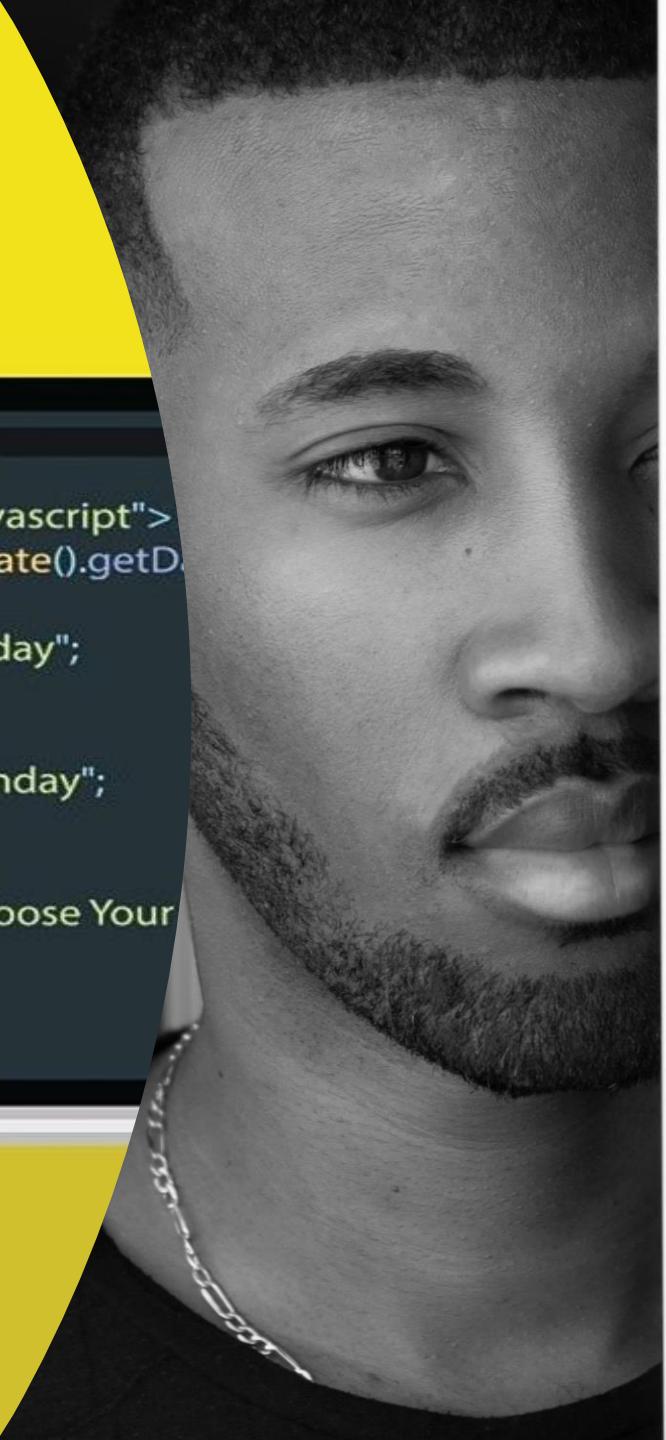


Formateur JavaScript

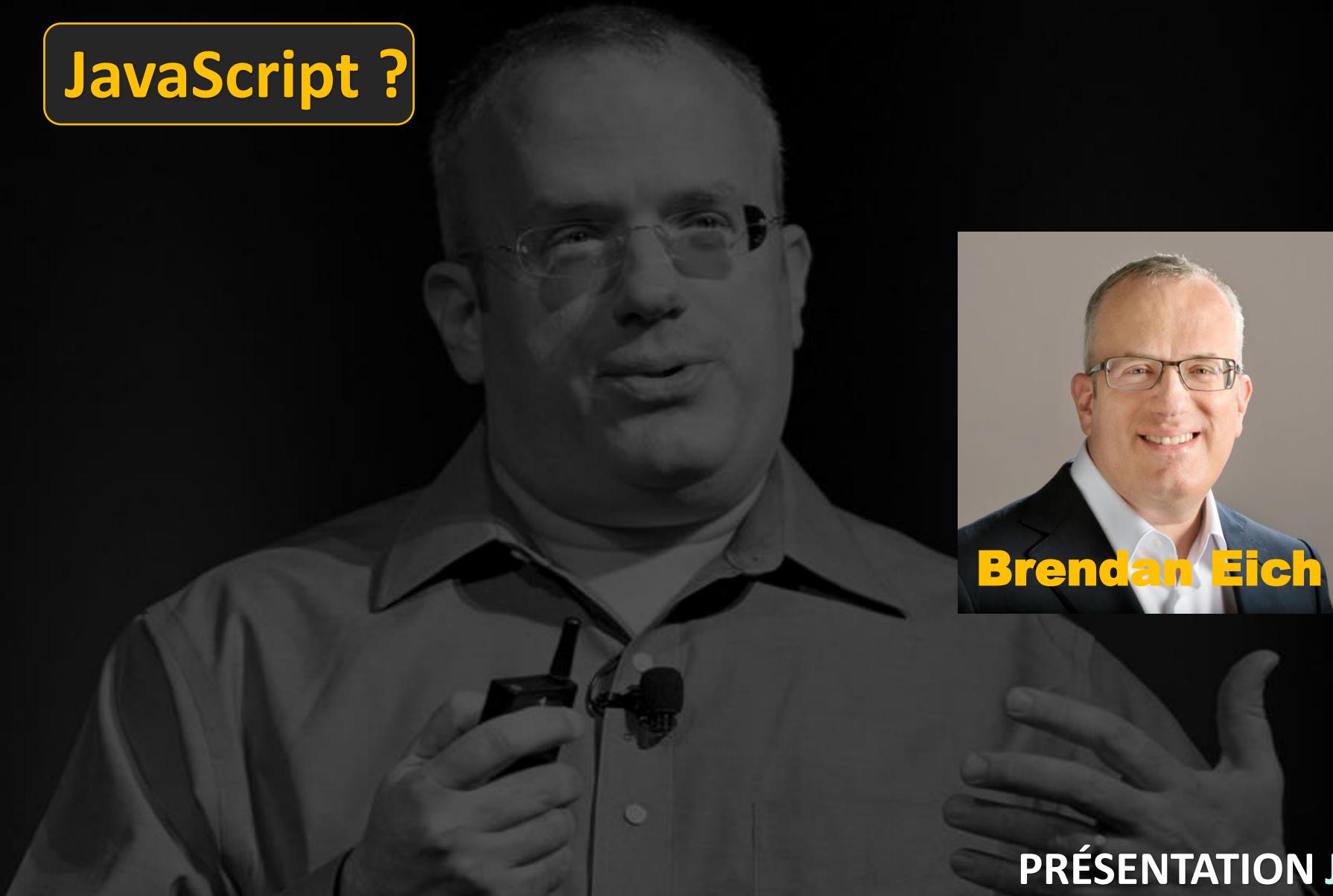
- Dostrel Jonathan
- Jonathan.dostrel@hotmail.com

```
<script type="text/javascript">
    switch (new Date().getDay()) {
        case 6:
            text = "Friday";
            break;
        case 0:
            text = "Sunday";
            break;
        default:
            text = "Choose Your Day!";
    }
</script>
```



Sommaire

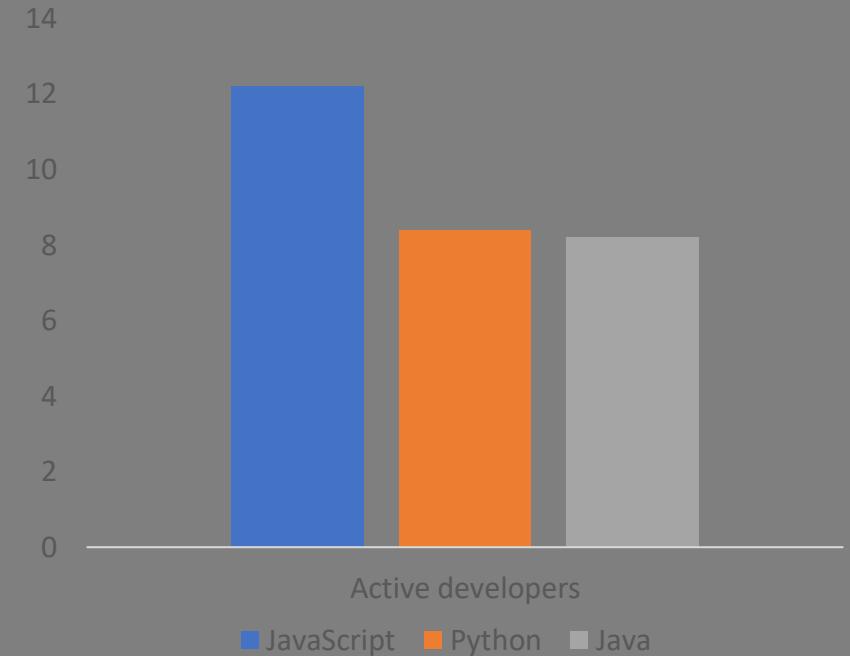
JavaScript ?



PRÉSENTATION JavaScript

Pourquoi vous avez cours de **JavaScript** ?

Résultat 2021



Java **Script**

La famille JavaScript

A quoi sert le

JavaScript ?



Structures



Mise en forme de
la Structures



Animation et
interaction

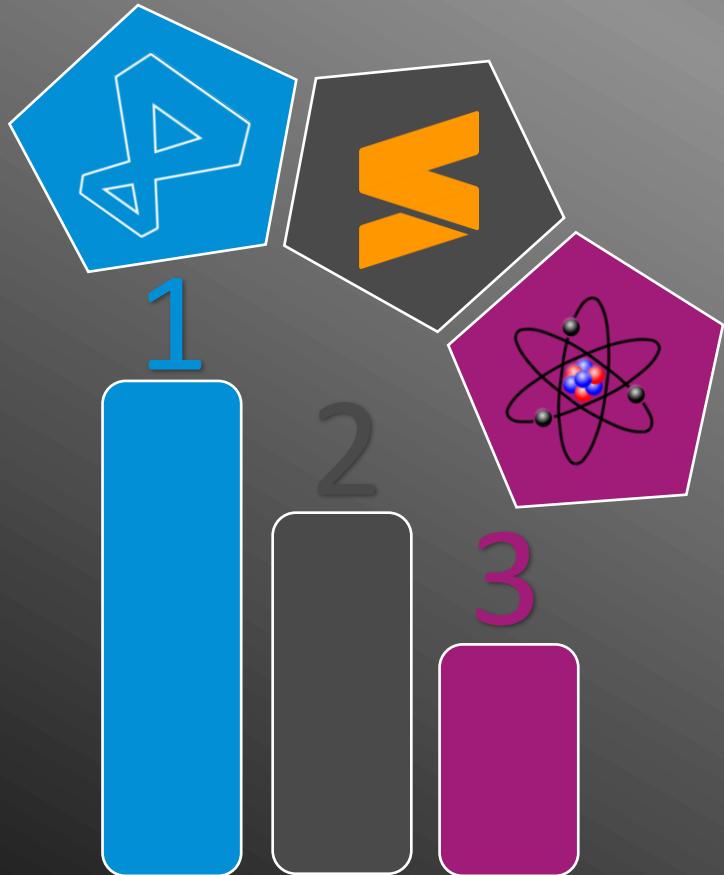
Exemple concret

iPhone 12 et iPhone 12 mini - Apple (FR)

Dom

```
↳ index.html > ⚒ html
1  <!DOCTYPE html>
2  <html lang="en">
3  <head>
4      <meta charset="UTF-8">
5      <meta http-equiv="X-UA-Compatible" content="IE=edge">
6      <link rel="stylesheet" href="./index.css">
7      <title>JavaScript</title>
8  </head>
9  <body>
10
11      <button id="Monbutton">Lancer</button>
12      <div id="text"></div>
13  </body>
14  <script src="./index.js" type="text/javascript"></script>
15  </html>
```

Outils nécessaires



Mozilla



Chrome



Safari

Console

Plan de cours

- Les instructions
- Les Variables
- Les Boucles
- Les conditions
- Les fonctions
- Dom
- Les évènements
- Les animations



- Facile à prendre en main
- Grand communauté
- performant
- Interaction

Les instructions

À quoi ressemble une instruction ?

Explication

Une instruction en javascript est un morceau de code, donnant un ordre à la machine et ponctué par un point-virgule.
console.log("Salut");

Qu'est-ce que la console ?

La console est une interface de votre navigateur, peu connue du grand public, et pour cause elle sert essentiellement aux développeurs. Ce que vous serez bientôt. (:
On l'utilise principalement pour débugger. Vous verrez tout cela en détail.

Explication

Comment ouvrir la console dépend du navigateur utilisé ?

En générale en appuyant sur F12

Dans le cas où cela ne fonctionne pas, faites un clique droit de la souris sur la page web ensuite cliquez sur inspecter

Les variables

Déclaration des variables



Explication

Une variable consiste en un espace de stockage, qui permet de garder en mémoire tout type de données. La variable est ensuite utilisée dans les scripts. Une variable contient seulement des caractères alphanumériques, le \$ (dollar) et le _ (underscore) ; elle ne peut pas commencer par un chiffre ni prendre le nom d'une fonction existante de Javascript. On crée la variable et on lui affecte (ou attribue) une valeur :

```
var numero;  
numero = 2  
var numero = 2;
```

C'est la plus connue, celle qui existe depuis Javascript 1.0. On peut résumer, même si c'est plus subtile que ça, que *var* est similaire à *let*, mais avec une portée de fonction (là où *let* a une portée de bloc).

```
let boolean;  
boolean = true / false  
let boolean = true / false
```

Sa principale caractéristique est sa portée : elle est limité à celle du bloc courant. Pour rappel, un bloc en Javascript, c'est ce qu'on retrouve entre accolades : une comparaison en *if*, une boucle *while* etc

Explication

```
const myString = 'text'
```

La portée de **const** est celle du bloc, comme la déclaration *let*. Pour la rendre globale, il faut simplement la définir hors de toute fonction

Comme **const** déclare une constante, on ne peut pas en re-déclarer une qui partage le même nom, dans la même portée :

Les variables(Exo)

Exercice sur les variables avec le formateur

Les conditions

if, else if, else, switch



Qu'est-ce qu'une structure de contrôle ?

Qui dit structure de contrôle dit donner une série d'instructions à la machine, et la machine exécute ces instructions dans l'ordre.

Une structure de contrôle est une instruction un peu spéciale qui permet de distordre cet espace-temps figé, d'en contrôler plus finement le flot.

Il existe plusieurs structures de contrôle en programmation. Les deux principales sont les boucles et les conditions.

Les boucles servent à dire « Voici une série d'instructions, exécute-les un certain nombre de fois d'affilée. »

Nous y reviendrons plus tard. Ce qui nous intéresse aujourd'hui, c'est la condition.

Qu'est-ce qu'une condition ?

Nous pouvons dire à un programme la chose suivante : « Voici une ou plusieurs instructions. Exécutez-les si et seulement si une certaine condition est remplie. »

Vous vous souvenez des booléens ?

C'est à cela qu'ils servent. C'est avec une expression booléenne, c'est-à-dire une expression ne pouvant évaluer qu'à true ou false, que nous pouvons conditionner l'exécution ou non d'une série d'instructions.

Résumé

Grâce à l'opérateur de comparaison `==`, nous avons vérifié la valeur de la variable `isUserAnonymous`. Ne vous inquiétez pas, nous arrivons tout de suite aux opérateurs de comparaison.

Ce qui est à retenir ici, c'est que nous avons pu conditionner une instruction.

```
const isUserAnonymous = true;  
  
if (isUserAnonymous === true) {  
    console.log("Salut !");  
} else {  
    console.log("Salut Magali !");  
}
```



Concrètement, nous avons dit :

Si l'utilisateur est anonyme, dis « Salut ! ».
Sinon, dis « Salut Magali ! ».

Les Comparaisons

`!==, ===, =, >, <, <=, >=`



Opérateurs de comparaison

Vous savez tous jouer à « Vrai ou faux »

« $3 + 4$ vaut **9**. – **Faux** ! »

« La Commune de Paris a eu lieu en 1871. – **Vrai** ! »

« Toutes les roses sont rouges. – **Faux** ! »

Votre ordinateur sait y jouer aussi. Il sait poser des questions et y répondre.

```
const x = 3 + 4;
```

```
console.log(x === 9);
```

Si vous exécutez ce code, vous verrez **false** s'afficher dans la console.

Soit la variable **x** égale à la somme de **3** et **4**.

Affiche si la variable **x** vaut **9** ou non.

Il affiche donc bien « non » par la valeur **false**. À l'inverse, l'expression **x === 7** aurait bien été évaluée à **true**.

Opérateurs de comparaison

Le symbole `==` est un opérateur de comparaison.

Il en existe quelques-uns, mais ils sont faciles à retenir car leur logique est très intuitive.

Dans tous les cas, vous allez à partir de maintenant vous en servir quotidiennement.

Le principe d'un opérateur de comparaison est donc de comparer deux valeurs – une à sa gauche, une à sa droite – et de transformer toute cette expression en une valeur booléenne.

L'expression « `2` est supérieur à `3` » devient la valeur « `Faux` ».

Opérateurs de comparaison

Égalité Stricte ===

Le symbole === compare en réalité deux choses : la **valeur** et le **type**.

En effet, l'expression `4 === "4"` sera évaluée à ?

false, car le nombre `4` n'est pas égal au caractère "`4`".

```
const x = "Salut" === "Salut !"
```

Ici `x` vaut **false** aussi.

Notez le point d'exclamation. Ces *Strings* ne sont pas identiques.

```
const x = 12 === 6 + 6 ?
```

Le nombre `12` est-il égal à la formule `6 + 6` ? Oui ! Ici `x` vaut **true**

Sachez que vous verrez aussi régulièrement l'emploi du symbole ==, Ceci est un opérateur de comparaison non strict, comparant la valeur mais pas le type.

Par exemple, `4 == "4"` vaut true car le javascript va ici pratiquer une conversion de type implicite

Opérateurs de comparaison

Différence `!==`

Le symbole `!==`, c'est l'exact inverse de `==`. Il renvoie true si les deux côtés de l'expression sont différents, et false s'ils sont égaux.

```
const x = 4 const y = 4; console.log(x !== y);
```

La console va ici afficher `false`, puisque `x` et `y` ne sont pas différents.

Comme pour le `==` et son équivalent non strict `!=`, le `!==` possède lui aussi une version non stricte `!=`.

De la même façon, l'opérateur `!=` tente de pratiquer une conversion intelligente pour comparer deux valeurs.

Opérateurs de comparaison

Supérieur >

Le symbole >, vous l'avez déjà vu en mathématiques, il dit « supérieur à ».

```
const x = 2 * 10;
```

```
const y = 60;
```

```
console.log(x > y);
```

Ce code affiche false, puisque x (20) n'est pas supérieur à y (60).

Inférieur <

Le symbole < est l'inverse du symbole >. Il dit « inférieur à ».

```
const x = 4; const y = 12;
```

```
console.log(x < y);
```

4 est bien inférieur à 12, la console affiche donc true.

Opérateurs de comparaison

Supérieur ou égal `>=`

Le symbole `>=` dit « supérieur ou égal à ».

```
const x = 5 + 5;
```

```
const y = 5 * 2;
```

```
console.log(x >= y);
```

Ici `x` et `y` valent 10, ils sont donc égaux et répondent positivement à la comparaison « supérieur ou égal à ».

La console affiche `true`.

Inférieur ou égal `<=`

Le symbole `<=` dit « inférieur ou égal à ».

```
const x = 5 * 3;
```

```
const y = 3 * 3;
```

```
console.log(x <= y);
```

Ici `x` vaut 15, et `y` vaut 9. 15 n'étant pas inférieur ou égal à 9, la console affiche `false`.

Les conditions

Partie 2



structure de contrôle IF

Nous avons vu ce qu'était une condition, nous avons vu comment comparer deux expressions pour les transformer en valeur booléenne, voyons donc comment faire combiner les deux pour déterminer si oui ou non on va exécuter un bloc d'instructions.

Nous créons une condition grâce au mot-clé **if**, suivi de parenthèses contenant l'expression à évaluer.

Si l'expression est vraie, on exécute les instructions contenues entre les accolades suivantes { ... }. Sinon, si l'expression est fausse, on exécute l'alternative optionnelle contenue dans le bloc **else** { ... }.

```
const userAge = 17;  
if (userAge >= 18) {  
    console.log("Vous êtes majeur, vous pouvez entrer.");  
} else {  
    console.log("Vous êtes mineur ? Dehors !");  
}
```

Si l'utilisateur a 18 ans ou plus, le programme exécute le bloc if. Sinon, il exécute le bloc else. Notez que le bloc else n'est pas obligatoire

structure de contrôle IF

```
const firstName = "Jacques";
console.log("Salut !");
if (firstName === "Magali") {
    console.log("Ah ! Mais c'est toi, je t'avais pas reconnue ! Quoi de neuf ?");
}
```

structure de contrôle IF

Notez aussi que nous ne sommes pas limités à deux blocs.

```
const x = 3;
if (x === 0) {
    console.log("x vaut 0.");
} else if (x === 1) {
    console.log("x vaut 1.");
} else if (x === 2) {
    console.log("x vaut 2.");
} else if (x === 3) {
    console.log("x vaut 3.");
} else {
    console.log("x ne vaut ni 0, ni 1, ni 2, ni 3.");
}
```

L'expression **else if** équivaut à « sinon si ». Traduisons le code précédent :

structure de contrôle IF

Tous les opérateurs de comparaison fonctionnent, puisque leur rôle est de transformer une expression en valeur booléenne, et qu'une condition attend précisément une valeur booléenne.

```
if (2 * 2 > 3) {  
    console.log("Tout va bien, 4 est toujours supérieur à 3.");  
}
```

De même, le code suivant, tout bête soit-il, fonctionne très bien :

```
if (true) {  
    console.log("Salut");  
}
```

Ce code affichera systématiquement "Salut" dans la console. Tandis que...

```
if (false) {  
    console.log("Salut");  
}
```

... ce code n'affichera jamais "Salut".

structure de contrôle IF

Petite note sur le point d'exclamation « ! »

Le symbole ! peut être utilisé dans une expression booléenne pour dire « n'est pas ». ! interroge l'inverse de l'assertion qui suit.

Par exemple :

```
console.log(!true)
```

Ce code affichera false.

```
const isUserAnonymous = false;  
if (!isUserAnonymous) {  
    console.log("Salut Magali !");  
}
```

« Si l'utilisateur n'est pas anonyme, salue-le avec son prénom. »

Y a-t-il d'autres formes de conditions ? Oui. Le switch et l'opérateur ternaire.

Les boucle

For, while, do while



Qu'est-ce qu'une boucle ?

Nous pouvons dire à la machine de répéter un bloc d'instructions.

« Dis bonjour dix fois d'affilée. »

« Demande à l'utilisateur son âge, encore et encore, tant qu'il n'a pas fourni un nombre valide. »

Le pauvre Jacques reçoit l'instruction suivante de son épouse : « Lave cette assiette ! »

C'est une instruction simple, il lave une assiette.

Mais si elle lui demande « Lave ces assiettes ! », au pluriel, il va laver une assiette, puis une assiette, puis une assiette.

Faire la vaisselle pourrait s'exprimer ainsi : « Voici une pile d'assiettes. Tant que la pile d'assiettes sales n'est pas vide, lave l'assiette suivante.

Quand tu as terminé la vaisselle, tu peux reprendre le cours de ta vie. »

L'action de laver une assiette est répétée un certain nombre de fois.

C'est une boucle.

Boucle while ?

il existe plusieurs boucles dans notre exemple nous allons prendre la boucle while

```
let age = window.prompt("Quel est votre âge ?");
```

```
while (isNaN(age)) {
    age = window.prompt("Merci de répondre un nombre. Quel est votre âge ?");
}
console.log("Vous avez " + age + " ans.");
```

Traduisons :

Demande son âge à l'utilisateur.

Si l'utilisateur n'a pas répondu un nombre, et tant qu'il n'aura pas répondu un nombre, continue indéfiniment de lui demander son âge.

Une fois qu'il aura bien répondu un nombre, tu peux reprendre le cours des instructions et utiliser ce nombre.

Boucle while ?

Mais que se passe-t-il si la condition ne devient jamais false ?

```
const age = 21;  
while (age > 18) {  
    console.log("Tomate");  
}  
console.log("Pomme de terre");
```

La variable age est fixée à 21.

Puis on dit « Tant que 21 est supérieur à 18, exécute cette instruction. »

Mais 21 sera absolument toujours supérieur à 18 ! Cette boucle ne s'arrêtera jamais.

On appelle cette situation une boucle infinie.

C'est un bug très courant qui fera planter votre navigateur, puisque celui-ci va faire mouliner le processeur à toute vitesse dans le vide.

On essaie d'éviter ! Ça vous arrivera de moins en moins souvent avec l'expérience, heureusement.

D'autre boucle que while ?

Oui. Nous avons les boucles *do while* et les boucles *for* qui sont aussi très utilisées. En vérité, *while*, *do while* et *for* sont les trois types de boucles principaux en programmation impérative. Ces boucles existent dans la grande majorité des langages impératifs, et constituent la base historique de l'algorithmie au même titre que les conditions *if*.

Et / OU
&& / ||

Et et Ou ?

Une expression booléenne peut être composée de plusieurs propositions.

Vous pouvez avoir plusieurs raisons de refuser l'entrée à une personne dans votre boîte de nuit : elle est habillée n'importe comment, **ou** elle est déjà connue de l'établissement pour provoquer des bagarres, ou sa tête ne vous revient pas.

Vous pouvez également avoir plusieurs conditions d'accès à une université : avoir eu au moins 14 au bac, **et** être âgé d'au moins 18 ans, **et** vous avez rempli un dossier d'inscription.

L'algorithmie permet également de fragmenter une condition avec des **et** et des **ou**.

Et et Ou ?

Le symbole `&&` permet de dire « et » dans une expression booléenne.

```
if (6 > 5 && 4 < 0) {  
    console.log("Salut");  
}
```

Si 6 est supérieur à 5, et si 4 est inférieur à 0, alors affiche « Salut ».

La première des deux assertions est vraie : 6 est bien supérieur à 5. Mais 4 n'étant pas inférieur à 0, c'est toute l'expression qui est alors considérée comme fausse.

Ou `||`

Le symbole `||` exprime « ou ».

Même exemple :

```
if (6 > 5 || 4 < 0) {  
    console.log("Salut");  
}
```

Si 6 est supérieur à 5, ou si 4 est inférieur à 0, alors affiche « Salut ».

4 n'est toujours pas inférieur à 0, mais 6 est encore supérieur à 5. À partir du moment où une seule des assertions est vraie, c'est toute l'expression qui est considérée comme vraie.

Les Tableaux

tableau

Tableau : Valeurs scalaires et collections ?

Jusqu'ici nous avons joué avec deux principaux types de valeurs : *Number* et *String*. Des nombres et des chaînes de caractères.

```
let x = 20;  
let y = "Jacques";
```

Dans la variable **x**, nous stockons la valeur 20. Dans la variable **y**, nous stockons la valeur "Jacques".

À chaque fois une valeur unique pour une adresse mémoire unique.

C'est ce qu'on appelle en informatique une « valeur scalaire ». En javascript, les nombres et les chaînes de caractères sont des types scalaires.

Mais stocker une valeur unique dans une variable, c'est comme les antibiotiques : c'est pas automatique.

Les langages de programmation nous offrent la possibilité de stocker plusieurs valeurs dans une seule et même variable.

On peut alors parler de « collection », ou plus rarement de « valeur composite ».

Tableau

La principale forme de collection dans le monde de la programmation est ce que nous appelons le « tableau », qu'on appelle aussi souvent sous son nom anglophone « *array* ».

Un tableau est la représentation centralisée d'un ensemble de valeurs.

En javascript, notez que l'on parle d'*Array* plutôt que d'*array*, avec une majuscule. De la même manière, nous parlons depuis le début de *Number* et de *String*. Cette distinction a son importance.

En javascript, les types sont en réalité des objets, et les types scalaires ne sont donc pas vraiment des types scalaires.

Si un tiroir est une adresse mémoire contenant une valeur unique, imaginez le tableau comme une armoire.

Vous pouvez toujours accéder à chaque tiroir individuellement, mais vous pouvez aussi manipuler tout le meuble en une fois si besoin.

Tableau

```
const users = ["Magali", "Jacques", "Josiane"];
```

La variable users ne contient pas une mais trois valeurs.

Voyez un tableau comme une adresse mémoire stockant des sous-adresses.

Au passage, les tableaux pouvant parfois être longs, n'hésitez pas à les *wrapper* sur plusieurs lignes ;

le javascript n'a aucun problème moral avec ceci :

```
const users = [  
    "Magali",  
    "Jacques",  
    "Josiane"  
];
```

Le tableau est délimité par des crochets [], et les valeurs à l'intérieur du tableau sont séparées par des virgules ,.

Tableau

Notez qu'un tableau peut ne contenir qu'une seule valeur si le cœur lui en dit :

```
const ages = [18];
```

Ou même aucune :

```
const apples = [];
```

Un tableau peut même contenir des tableaux :

```
const garden = [
  [
    ["Cornue des Andes", "Cœur de bœuf"],
    ["Charlotte", "Mona Lisa"]
  ],
  [
    ["Verte de Milan", "Patidou"],
    ["Petit Gris de Rennes", "Noir des Carmes"]
  ],
];
```

Aucun problème avec un tableau contenant des sous-tableaux, lesquels peuvent eux-mêmes contenir des sous-tableaux, et ainsi de suite. On parle de tableaux à plusieurs dimensions.

Le tableau à deux dimensions est d'ailleurs très répandu, c'est le principe d'une matrice.

Et notre exemple précédent, garden, est un tableau à trois dimensions.

Comment accéder aux valeurs ?

C'est beau une armoire, mais encore faut-il pouvoir accéder individuellement à chaque tiroir.

```
const users = ["Josiane", "Jacques"];
const willWashDishes = users[1];
```

C'est simple. Nous avons stocké deux utilisateurs dans la variable users au moyen d'un tableau.
À ce moment, ligne 1, le javascript a attribué une « clé numérique » à chaque entrée du tableau.
Les clés sont attribuées dans l'ordre numérique.

Pour accéder à une entrée précise du tableau, on appelle la variable, suivie des crochets dans lesquelles on spécifie la clé de la valeur désirée : users[1].

Alors ? Qui est users[1] dans notre exemple ? Qui va faire la vaisselle ?

Josiane ?

Erreur ! "Josiane" n'est pas l'entrée numéro 1 mais l'entrée numéro 0. (:

Comment accéder aux valeurs ?

```
const fruits = ["Pomme", "Poire", "Banane"];
console.log(fruits[0]);
Ce morceau de code va afficher "Pomme".
```

```
const numbers = [2, 5, 22];
console.log(numbers[2]);
Ce morceau de code va afficher 22.
```

Et pour les tableaux imbriqués ? Rien de compliqué, c'est logique :

```
const numbers = [
    [1, 3, 5, 7],
    [2, 4, 6, 8]
];
console.log(numbers[0][3]);
Ce morceau de code va afficher 7.
```

Comment accéder aux valeurs ?

On pourrait aussi stocker nos sous-tableaux dans des variables :

```
const numbers = [  
    [1, 3, 5, 7], [2, 4, 6, 8]
```

```
];
```

```
const oddNumbers = numbers[0];  
const evenNumbers = numbers[1];
```

```
console.log(oddNumbers[2]);
```

Puisque la variable oddNumbers stocke désormais une copie du tableau trouvé à l'adresse numbers[0], ce code affiche 5.

Ajouter des éléments à un tableau

Comme ceci :

```
const users = ["Magali", "Jacques"];
users.push("Josiane");
console.log(users[2]);
```

Nous verrons plus en détail le principe des fonctions et des méthodes.

Mais pour l'heure, retenez que grâce à la méthode **push** à laquelle on a passé l'argument "Josiane", on a pu ajouter un élément au tableau de départ.

Celui-ci ressemble désormais à ça : ["Magali", "Jacques", "Josiane"].

L'élément est ajouté à la fin du tableau.

Attends, attends ! On n'avait pas dit que le mot-clé const empêchait de modifier une variable précédemment déclarée ?

Oui, on a dit ça. Mais on a aussi dit dans une petite parenthèse de ce type qu'un *Array* en javascript était en vérité un objet. Et les objets, question immutabilité, c'est une autre histoire. Vous n'y êtes pas encore. Retenez pour l'instant que vous pouvez modifier le contenu d'un tableau javascript même s'il a été déclaré avec const. Ce que vous ne pouvez pas faire, par contre, c'est écraser la variable par une autre valeur que ce tableau.

Les tableaux c'est super !

Ils peuvent vous sembler un peu abstraits et inutiles pour l'instant.

Tout cela peut vous sembler très théorique et pompeux. Pourquoi ne pas stocker directement nos valeurs dans des variables ?

Nous allons très vite voir des cas pratiques, notamment grâce au principe d'itération.

Boucle For

For



Boucle for !

Souvenez-vous de la boucle while.

```
let msg = window.prompt("Dites « Bonjour » :");
while (msg !== "Bonjour") {
    msg = window.prompt("Dites « Bonjour » :");
}
document.write(msg);
```

Tant que l'utilisateur n'a pas dit bonjour, continue à lui demander indéfiniment de dire bonjour.

Cette boucle très simple fonctionne autour d'une expression booléenne, et continue de boucler tant que celle-ci est vraie.

Boucle for !

Exemple :

```
for (let i = 0; i < 5; ++i) {  
    document.write(i.toString());  
}
```

Mais avant de pouvoir la comprendre, attardons-nous sur deux notions : le *scope*, et l'incrémentation.

Portée des variables, ou *scope*

Nous parlions au début de cette formation du mot-clé var.

Nous disions qu'il s'agissait de la façon historique de déclarer une variable, mais qu'il ne fallait pas l'utiliser pour des raisons que nous verrions plus tard.

Ce plus tard est arrivé.

Quand vous construisez une structure de contrôle, vous enclavez une ou plusieurs instructions dans un « bloc ».

Boucle for !

Exemple :

```
if (true) {  
    let msg = "Salut";  
} console.log(msg);
```

Ce code ne fonctionnera pas. Pourquoi ? Parce que j'ai déclaré une variable msg dans un if.

Ce if crée un contexte particulier à l'intérieur de ses accolades, et les variables let et const déclarées dans ce contexte ne sont pas accessibles en dehors.

```
let msg = "Salut";  
if (true) { console.log(msg); }
```

Et ça, ça fonctionne ? Oui ! Mais pourquoi ?

Parce que cette fois, la variable msg est déclarée en amont dans le *scope* parent. Il y a une hiérarchie entre les *scopes*. Vous pouvez créer des *scopes* dans des *scopes* dans des *scopes*, et la portée des variables y descend en cascade, mais elle ne remonte jamais.

Et var, dans tout ça ? var n'obéit pas à cette règle désormais sacrée. Les variables déclarées avec var sont capables de remonter la cascade des *scopes*, comme des truites. Et c'est pas bien ? C'est pratique, non ? Non, pas vraiment. Le principe de cascade est un schéma récurrent en programmation, vous l'utilisez même en CSS. Plus vous prendrez d'expérience, plus vous comprendrez pourquoi il est bon d'éviter var.

Incrémantion !

Il est parfois utile d'ajouter 1 à un nombre.

Comment feriez-vous ?

```
let x = 5;  
x = x + 1;
```

```
console.log(x);
```

Ce code affichera 6. C'est une façon de faire.

Il en existe une plus courte :

```
let x = 5;
```

```
x += 1;
```

```
console.log(x);
```

Même résultat.

Encore plus court :

```
let x = 5;
```

```
++x;
```

```
console.log(x);
```



Incrémantion !

On appelle cela l'incrémantion. Les signes `++`, puis le nom de la variable, modifient celle-ci pour lui ajouter 1.

Il va sans dire que cette variable doit donc contenir un nombre.

Notez que `++x` et `x++` fonctionnent aussi bien l'un et l'autre pour notre usage. Il s'agit respectivement de pré-incrémantion et de post-incrémantion.

La pré-incrémantion incrémente d'abord puis retourne la valeur, et à l'inverse la post-incrémantion retourne la valeur puis incrémente.

En pratique :

```
let x = 5;  
console.log(x++);  
console.log(x);  
let y = 5;  
console.log(++y);  
console.log(y);
```

Les trois premières lignes afficheront 5 puis 6, et les trois dernières afficheront 6 puis 6.

Décrémentation !

```
let x = 5; --x;  
console.log(x);  
Ce code affiche 4
```

Retour au for !

Nous sommes désormais armés pour comprendre une boucle for classique.

Reprendons notre premier exemple :

```
for (let i = 0; i < 5; ++i) {  
    document.write(i.toString());  
}
```

Les parenthèses de la boucle for sont divisées en trois sections séparées par des points-virgules ;. Oui, c'est un peu bizarre.

La première section sert à déclarer une ou plusieurs variables.

Ici nous déclarons une variable let i = 0.

Si nous avions voulu en déclarer plusieurs, nous les aurions séparées par des virgules , : let i = 0, j = 1.

Notez que cette façon de déclarer plusieurs variables en même temps fonctionne même dans le reste de votre code, pas seulement dans les boucles for.

Mais cela pourrait rendre votre code moins lisible. En dehors des boucles for, préférons déclarer une variable par ligne.

Retour au for !

La deuxième section sert à déclarer la condition de notre boucle.

Et cette deuxième section fonctionne exactement comme la condition de la boucle while.

Toute boucle a besoin d'une condition pour savoir si elle doit continuer à se répéter ou non.

Tant que l'expression booléenne de cette deuxième section est vraie, la boucle for se répète.

La troisième section sert à effectuer une action à chaque nouveau tour de boucle.

Insistons sur « nouveau » : cette action ne s'exécute pas au premier passage.

Que va donc produire notre exemple ?

Afficher 0, puis 1, puis 2, puis 3, puis 4, puis... plus rien. Pas 5 ? Non, pas 5.

Nous disons à la boucle « Continue de tourner tant que i est inférieure à 5. Dès que cette variable devient supérieure ou égale à 5, la boucle s'arrête et le code reprend son cours.

C'est un peu alambiqué, non ? À quoi ça peut bien servir, une boucle for ? Sa puissance se révèle quand on la combine aux tableaux.

Retour au for !

Voyons à quoi peut ressembler la combo **for** + **array** :

```
const heroes = ["Buffy", "Xander", "Willow"];  
  
const numberOfHeroes = heroes.length;  
for (let i = 0; i < numberOfHeroes; ++i) {  
    console.log(heroes[i]);  
}
```

Nous faisons ici ce que l'on appelle « parcourir un tableau ».

Ligne 1, on déclare un tableau contenant trois éléments.

Ligne 2, si vous ne connaissiez pas encore, c'est très simple : la propriété length d'un tableau renvoie le nombre d'éléments du tableau.

heroes.length renvoie ici 3.

Puis intervient notre boucle for qui va afficher tour à tour chaque élément du tableau. Traduisons ce morceau de code en français :

Retour au for !

```
const heroes = ["Buffy", "Xander", "Willow"];
```

```
const numberOfHeroes = heroes.length;  
for (let i = 0; i < numberOfHeroes; ++i) {  
    console.log(heroes[i]);  
}
```

La boucle va exécuter son bloc d'instructions une première fois.

Cette première fois, i vaut 0, on affiche donc heroes[0], c'est-à-dire "Buffy".

Puis on fait un deuxième tour ; cette fois on exécute ++i, i vaut donc 1 et heroes[1] va afficher "Xander".

Puis un troisième tour de boucle, ++i amène i à la valeur 2, heroes[2] affiche "Willow".

Puis encore une fois ++i, i vaut cette fois 3, et... stop ! Coupez ! La variable i n'est plus inférieure au nombre de héros (3), la condition d'exécution de la boucle n'est donc plus remplie et on reprend le flux normal du code.

Les types

String, Number, Boolean, Array



Ces sales le type !

On en est où, question types ?

- Number
- String
- Boolean
- Array

Trois types scalaires, et une collection.

Null est un type particulier.

Vous savez que le type booléen ne peut avoir que deux valeurs possibles, false et true.

Le type null, lui, est encore plus simple, il ne peut contenir que la valeur null, qui représente en fait virtuellement l'absence de valeur.**(c'est une valeur pour dire « pas de valeur ».)**

Ces sales le type !

```
const userAge = window.prompt("Quel est votre âge ?");
let msg;
if (Number(userAge) >= 18) {
    msg = "Tu es majeur ! Bienvenue."
} else {
    msg = null;
}
if (msg !== null) {
    document.write(msg);
}
```

Vous utiliserez régulièrement null pour représenter une valeur vide. C'est nécessaire à la logique algorithmique.

que vaut la variable msg ?

undefined

```
let msg; console.log(msg);
```

Ce code affiche undefined.

La variable existe, et elle est d'un type très propre au javascript. Comme null, le type undefined ne peut s'exprimer que sous une seule valeur : undefined. « Indéfini », en français.



Ces sales le type !

Quelques conversions

Nous avons déjà parlé de conversion de types.

Il s'agit de prendre une valeur d'un type donné, et d'essayer de la convertir vers un autre type.

Pourquoi essayer ? Parce que toutes les valeurs ne sont pas compatibles.

Vous pouvez convertir une branche d'arbre en bâton de marche, vous ne pouvez pas convertir une hache de bûcheron en hélicoptère.

```
const x = "5";  
const y = Number(x);
```

La chaîne "5" peut facilement être convertie en nombre 5. Et à l'inverse :

```
const x = 5;  
const y = String(x);
```

Le nombre 5 peut être converti en la chaîne "5". Mais nous avons vu que toutes les valeurs ne sont pas compatibles :

```
const x = "Salut";  
const y = Number(x);
```

Ici y vaut NaN, pour Not a Number. Demander à la machine de transformer « Salut » en un nombre n'a aucun sens, elle ne comprend pas ce qu'on lui ordonne et répond que la valeur qu'on lui a fournie n'est pas un nombre.

Ces sales le type !

Plutôt qu'utiliser Number, nous pouvons forcer un nombre à être soit entier soit décimal en utilisant les fonctions parseInt et parseFloat, que vous verrez aussi très souvent :

```
const x = "5.5";
const y = parseInt(x);
Ici y vaut 5.5.
```

Alors que le code suivant...

```
const x = "5";
const y = parseFloat(x);
... transforme au contraire un nombre entier en nombre « flottant » y vaut 5.0.
```

Les fonctions

Qu'est-ce que une fonction !

Si je vous demande de résoudre $2 + 2$, vous me répondez – je l'espère – 4.

Qu'est-ce que vous venez de faire ?

Vous avez pris un premier nombre, dans votre tête vous lui avez ajouté un second nombre, et enfin vous m'avez donné votre réponse. C'est une **fonction**.

Une fonction très simple, qui procède à une unique opération arithmétique, mais une fonction quand même.

Dès vos premières années votre cerveau a appris à additionner des nombres.

Il a fallu vous lui expliquer le principe un jour, mais depuis il sait le faire sans y penser.

Et quand on lui demande d'additionner 6 à 9, il répondra 15 sans problème.

Votre cerveau a mémorisé un protocole précis, et est capable de le reproduire à volonté quand on le lui demande.

Qu'est-ce que une fonction !

En javascript, une fonction servant à additionner deux nombres pourrait ressembler à cela :

```
function add(x, y) {  
    const sum = x + y;  
    return sum;  
}  
const result = add(4, 7);  
console.log(result);
```

on appelle la fonction add précédemment déclarée avec les paramètres 4 et 7, et on stocke le « retour » de cet appel dans une variable.
on affiche le résultat : 11.

Qu'est-ce que une fonction !

Détaillons tout cela.

Le mot-clé function dit au javascript que ce qui suit est la déclaration d'une fonction.

Ensuite vient add après un espace.

C'est un nom que le développeur a arbitrairement choisi pour essayer d'exprimer le plus clairement possible le rôle de cette fonction.

C'est en général un verbe représentant l'action exécutée par la fonction.

Comme pour les variables, on respecte la notation « camelCase », et on s'efforce d'être à la fois clair et concis.

Ensuite nous avons des parenthèses (x, y).

Les parenthèses, après le nom de la fonction, sont là pour déterminer ce que nous appelons les « paramètres », ou encore « arguments ».

Les paramètres sont des données d'entrée nécessaires à l'exécution de la fonction.

En effet, si on écrit une fonction dont le rôle est d'additionner un nombre à un autre nombre, le développeur doit pouvoir passer ces deux nombres à ladite fonction quand il appelle celle-ci.



Qu'est-ce que une fonction !

Le nombre d'arguments peut être aussi élevé que nécessaire, et les arguments sont séparés par une virgule.

Il peut aussi n'y avoir qu'un argument : `displayMessage(msg)`.

Puis, comme pour les structures de contrôle que sont les conditions et les boucles, on encadre un bloc d'instructions entre des accolades {}.

C'est le code de la fonction à proprement parler.

Ce code sera exécuté à chaque fois que la fonction sera appelée.

Si la fonction déclare des arguments, ils sont accessibles au sein de ce bloc d'instructions en tant que variables, dont les valeurs sont celles que l'on aura passées en appelant la fonction.

Après avoir déclaré ma fonction `add(x, y)`, je l'appellerai sous la forme `add(4, 5)`.

Le code de la fonction déclarée précédemment s'exécutera en assignant 4 à la variable x et 5 à la variable y.

Qu'est-ce que une fonction !

Et à la fin de notre bloc d'instruction, le mot-clé `return` sert à « renvoyer » une valeur.

En somme, quand on appelle une fonction, elle renvoie la valeur qui suit `return`.

Le mot-clé `return` stoppe complètement la fonction, et tout ce qui se trouverait après ne sera pas exécuté.

Notez toutefois que `return` n'est pas obligatoire.

Une fonction peut parfaitement se contenter d'exécuter une action sans avoir besoin de renvoyer une valeur.

Par exemple :

```
function displayMessage(msg) {  
    console.log(msg);  
}  
  
displayMessage("Salut");  
displayMessage("la");  
displayMessage("Planète");
```

```
function add(x, y) {  
    const sum = x + y;  
  
    return sum;  
}  
  
const result = add(4, 7);  
console.log(result);
```

DRY - Don't Repeat Yourself !

La programmation c'est beaucoup de conventions et de principes.

On vise un code propre et élégant.

Si on vous demande de fabriquer une porte mais que vous mettez la serrure tout en haut ou les gonds tout en bas, votre porte sera à peu près fonctionnelle mais très peu pratique.

Les gens perdront beaucoup de temps à comprendre comment elle s'ouvre, voire se blesseront ou abîmeront le sol.

Le code c'est pareil, il y a beaucoup de façons de parvenir à un résultat, mais on essaie de se rapprocher de la plus intuitive, simple et élégante.

Cette philosophie, en programmation, passe entre autres par le principe DRY.

DRY - Don't Repeat Yourself !

Considérons le code suivant, visant à retirer les 21 % de TVA des prix des ventes effectuées hier et aujourd'hui :

```
const soldYesterday = [20, 40.55, 3.12, 99.95];
const soldToday = [15, 45.55, 13.55, 130.99];

for (let i = 0; i < soldYesterday.length; ++i) {
    soldYesterday[i] = soldYesterday[i] - soldYesterday[i] * 21 / 100;
}
for (let i = 0; i < soldToday.length; ++i) {
    soldToday[i] = soldToday[i] - soldToday[i] * 21 / 100;
}
```

Outch ! On répète quasiment le même code. Que se passe-t-il si on doit non plus traiter deux jours mais une semaine ouvrée entière ? On devra répéter cette boucle cinq fois, avec des noms de variables de plus en plus confus. Et quand la TVA passera à 22 % ? Elle est cachée à cinq endroits différents dans des calculs qui se répètent, la changer partout amènera le risque d'en oublier une occurrence et donc de produire un bug sournois.

DRY - Don't Repeat Yourself !

Corrigeons le tir :

```
const soldYesterday = [20, 40.55, 3.12, 99.95];
const soldToday = [15, 45.55, 13.55, 130.99];
const vatRate = 21;
```

```
function removeVatFromPrices(prices, vatRate) {
    for (let i = 0; i < prices.length; ++i) {
        prices[i] = prices[i] - prices[i] * vatRate / 100;
    }
}
removeVatFromPrices(soldYesterday, vatRate);
removeVatFromPrices(soldToday, vatRate);
```

Nous déclarons une fonction qui prend en premier argument un tableau de prix, et en second argument un taux de TVA exprimé en pourcentage.

Le taux de TVA de 21 est stocké dans une unique variable vatRate qui est ensuite utilisée plusieurs fois, ce qui n'est pas directement lié à la notion de fonction mais contribue tout de même au principe DRY.

Dès qu'un morceau d'algorithme se trouve utilisé à plusieurs endroits de votre code, et ça arrive tout le temps, il devient nécessaire de créer des fonctions afin de simplifier les choses.

Votre code devient plus facile à lire et à comprendre, que ce soit pour vos collègues ou même pour vous quand vous y reviendrez des semaines voire des mois plus tard.

Tout morceau de code appelé au moins deux fois gagne à se trouver enclavé dans une fonction réutilisable.

Et parfois, même si notre morceau de code n'est appelé qu'une fois, une fonction peut quand même aider à rendre votre programme plus clair et mieux organisé.

Valeur par défaut !

```
function displayMessage(n, msg = "Salut") {  
    for (let i = 0; i < n; ++i) {  
        console.log(msg)  
    }  
}  
displayMessage(2, "Hey");  
displayMessage(3);
```

Nous avons ici une fonction dont le rôle est d'afficher un certain message un certain nombre de fois.

Pour que cette fonction... fonctionne... elle a absolument besoin de l'argument `n`, qui spécifie le nombre d'itérations.

Concernant l'argument `msg`, cependant, une valeur par défaut "Salut" est définie.

Cela signifie que si, à l'appel de la fonction, on ne passe pas de valeur en lieu et place de l'argument `msg`, alors la valeur par défaut "Salut" sera utilisée.

Ce code affiche donc deux fois "Hey", puis trois fois "Salut".

Récursivité !

Une fonction peut s'appeler elle-même.

Sachez que c'est une chose qui existe, même si l'utilité d'un tel procédé vous échappe pour l'instant.

Prenons la fameuse suite de Fibonacci :

```
function fibonacci(x) {  
    if (x <= 1) {  
        return 1;  
    }  
    return fibonacci(x - 1) + fibonacci(x - 2);  
}
```

Ce qu'il faut retenir ici, c'est que notre fonction fibonacci s'appelle elle-même au sein de son propre code.

C'est ce qu'on appelle une fonction récursive.

Et comme pour une boucle, si une fonction récursive n'est pas pensée pour pouvoir arrêter de boucler, elle continuera à s'appeler infiniment et fera planter le navigateur.

Vous utiliserez assez peu la récursivité à vos débuts.

C'est une gymnastique mentale qui demande une certaine expérience en algorithmie, et dont l'usage demeure assez anecdotique en programmation impérative.

C'est une pratique plus propre à la programmation dite fonctionnelle, dans des langages tels que le *haskell* où la récursivité prend même la place des boucles.

Git

Versioning !

Git est un outil originellement créé par Linus Torvalds dans le but de gérer les versions des logiciels. Quand vous téléchargez la version v12.0.14 de votre jeu-vidéo préféré, il y a fort à parier en 2020 que derrière cette notation vous trouverez Git.

Git n'est pas le seul logiciel de gestion de versions au monde.

Vous pourriez entendre parler de CVS ou SVN. Mais en vérité, ces alternatives âgées n'arrivent pas à la cheville de la puissance de Git.

C'est pourquoi ce dernier s'est imposé mondialement dans le monde de la programmation, et vous l'utiliserez quotidiennement dans votre vie professionnelle. Même ces supports de cours sont maintenus grâce à Git, et imposent son utilisation régulière.

Quand je termine la rédaction d'un module, je le *commit* avec Git. Quand je trouve une erreur dans un cours plus ancien, je *commit* cette correction avec Git.

Git est notre filet de sécurité, il est utile même pour ce type de travail qui n'est pas strictement du code.

Chaque version stable d'un projet reçoit un numéro de version. Et chaque micro-modification du projet fait l'objet de ce qu'on appelle un *commit*.

Si je visualise le *commit* qu'un développeur aurait appelé Add money formatting function, j'y verrai logiquement l'ajout d'une fonction dont le rôle serait de formater l'affichage de monnaie.

Le *commit* Fix typo in README contiendrait une minuscule correction de faute d'orthographe.

Et un projet, pour Git, c'est une succession de *commits* qui forme un historique.

Versioning !

Puisque absolument chaque modification sur un projet fait l'objet d'un *commit*, en rejouant tous les *commits* dans l'ordre Git est capable de reconstituer de zéro l'état actuel du projet.

Imaginez une machine temporelle où, régulièrement, vous iriez appuyer sur un bouton *commit*. À chaque action dans votre vie, vous appuyez sur *commit*.

Vous faites une pâte à pizza le 23 juillet à 22:34:00, *commit*.

Vous sortez le chien le 25 juillet à 15:30:15, *commit*.

Nous sommes le 26 juillet, vous vous appercevez que votre décision prise le 24 juillet d'envoyer cette lettre à cette personne a causé une guerre mondiale.

Vous allez voir la machine temporelle, et vous modifiez le *commit* pour faire comme si cette lettre n'avait jamais été envoyée.

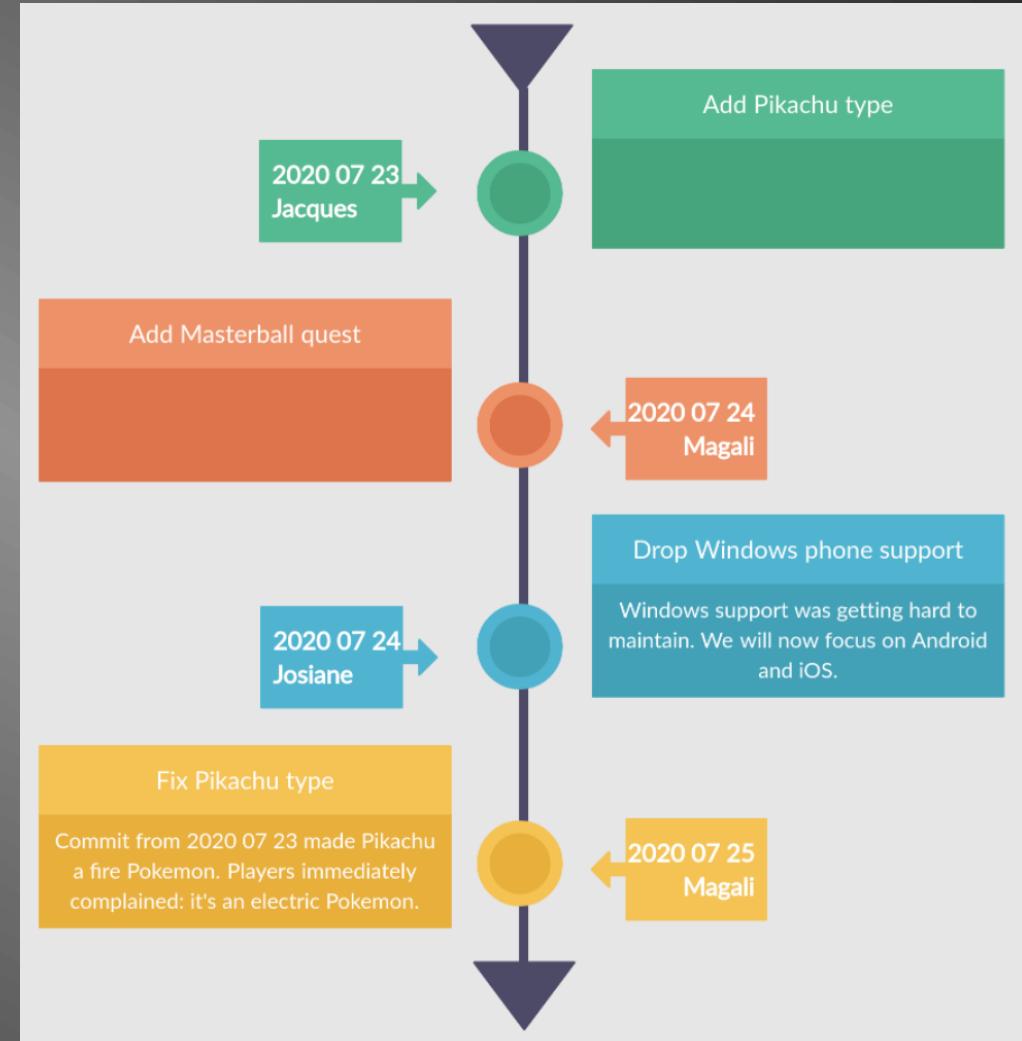
Vous réécrivez l'Histoire. Pas de guerre mondiale.

C'est ce que permet Git avec un projet informatique. Il vous aide à écrire petit à petit l'histoire de votre projet, et vous permet à tout moment de revenir sur cette histoire pour la comprendre ou même la modifier.

Versioning !

Le 23 juillet, Jacques ajoute le type de Pikachu. Il appelle ce commit [Add Pikachu type](#).

D'autres choses se passent ensuite, mais le 25 juillet, on voit que Magali a dû revenir le code de Jacques. Pourquoi ? Jacques s'était visiblement trompé, il avait fait de Pikachu un Pokemon de type feu. Magali corrige le souci et commit ce *fix* avec quelques explications sur le contexte.



Versioning !

Il existe énormément de documentations et de tutoriaux en ligne autour de Git.

En effet, c'est un outil indispensable mais aussi relativement difficile d'accès. On ne devrait pas dire ça dans un cours, mais faisons une petite exception et soyons honnêtes : Git, c'est complexe.

Et aucun cours théorique ne pourra prétendre remplacer des années de pratique.

Au terme de cette formation, vous ne serez pas des experts de Git, loin de là.

Notre objectif est avant tout de vous faire suffisamment travailler les bases pour que vous puissiez collaborer avec des professionnels sans avoir à rougir.

Ceux-ci n'auront pas tout à vous apprendre de zéro, et pourront alors prendre du temps pour vous aider à vous perfectionner et à compléter vos connaissances sur le tas.

Commandes Git basique !

git init :

La commande git init crée un nouveau dépôt Git.

Elle permet de convertir un projet existant, sans version en un dépôt Git ou d'initialiser un nouveau dépôt vide.

La plupart des autres commandes Git ne sont pas disponibles hors d'un dépôt initialisé, il s'agit donc généralement de la première commande que vous exécuterez dans un nouveau projet.

L'exécution de git init créer un sous-répertoire .git dans le répertoire de travail actif, qui contient toutes les métadonnées Git nécessaires pour le nouveau dépôt. Ces métadonnées incluent des sous-répertoires pour des objets, des réfs et des fichiers de modèle. Un fichier HEAD est également créé et pointe vers le commit actuellement extrait.

git clone

Petite remarque : git init et git clone sont facilement confondues.

À un niveau supérieur, les deux commandes permettent d'« initialiser un nouveau dépôt Git ». Toutefois, git clone dépend de git init. git clone permet de créer une copie d'un dépôt existant.

En interne, git clone appelle d'abord git init pour créer un dépôt.

Elle copie ensuite les données du dépôt existant, puis fait un check-out d'un nouvel ensemble de fichiers de travail. Découvrez-en plus sur la page [git clone](#).

git add

git add est un commandement important - sans lui, personne ne ferait jamais rien.

Parfois, peut avoir la réputation d'être une étape inutile dans le développement. Mais en réalité, c'est un outil important et puissant. vous permet de façonner l'histoire sans changer votre façon de travailler.git

Commandes Git basique !

git commit :

enregistre toutes les modifications intermédiaires, ainsi qu'une brève description de l'utilisateur, dans un « commit » dans le référentiel local.

Les commits sont au cœur de l'utilisation de Git.

Vous pouvez considérer une validation comme un instantané de votre projet, où une nouvelle version de ce projet est créée dans le référentiel actuel.

Enregistrer les modifications apportées au référentiel.
journal de bords.

git checkout

La commande git checkout vous permet de **basculer entre les branches créées au moyen de git branch**. Le check-out d'une branche entraîne une mise à jour des fichiers contenus dans le répertoire de travail, qui s'alignent sur la version stockée dans cette branche.

git branch

Si est donné, ou s'il n'y a pas d'arguments de non-option, les branches existantes sont répertoriées; la branche actuelle sera surlignée en vert et marquée d'un astérisque.

Toutes les branches vérifiées dans les arbres de travail liés seront mises en évidence en cyan et marquées d'un signe plus.

Commandes Git basique !

git push

Met à jour les références distantes à l'aide de références locales, tout en envoyant les objets nécessaires pour compléter les références données.

git fetch

La commande git fetch **télécharge les commits, les fichiers et les refs d'un référentiel distant dans votre référentiel local**. Aller chercher est ce que vous faites quand vous voulez voir ce sur quoi tout le monde a travaillé.

git merge

git merge combinera plusieurs séquences de commits en un historique unifié.

Dans les cas d'usage les plus fréquents, git merge est utilisée pour combiner deux branches.

[git merge | Atlassian Git Tutorial](#)

git rebase

La commande git rebase vous permet de modifier facilement **une série de commits, en modifiant l'historique de votre référentiel**. Vous pouvez réorganiser, modifier ou écraser les commits ensemble.

Commandes Git basique !

git rm

La commande git rm peut être utilisée pour **supprimer des fichiers individuels ou une série de fichiers**. git rm a pour fonction principale de supprimer les fichiers suivis de l'index Git.

En outre, la commande git rm permet de supprimer des fichiers de l'index de staging et du répertoire de travail.

[git rm | Atlassian Git Tutorial](#)

Projet Libre

Introduction :

Jusqu'à présent nous avons vu énormément de choses Et maintenant il est temps de mettre en place tout ce que j'ai appris

Ce que j'attends de vous :

Mettez-vous en groupe de 2 ou 3 Selon les préférences des apprenants afin de réfléchir à un mini projet sur une journée

Attention !

ne poussez pas le projet trop loin par manque de temps.

il est préférable, que par groupe il y a au moins une personne qui sache utiliser un peu Git

Objectif :

En vous aidant de l'objet fourni créer un mini site.

Faites exploser votre imagination afin d'avoir un rendu propre et agréable à voir

Vous devez aussi utiliser Git afin de récupérer le dépôt de base et aussi faire 2 commits Par personne En fonction de votre avancée.

Bonus:

Mettre en place un input qui nous permettra de filtrer nos images Crée la fonction qui va avec

Les Sélecteurs



Document Object Model

Jusqu'à maintenant, la découverte de JavaScript n'a pas été très utile pour nous permettre d'agir sur les pages web.

L'intérêt du javascript étant de transformer des pages **HTML** statique en applications web dynamiques, nous allons voir différentes manières d'interagir avec le dôme, plus spécifiquement différentes méthodes permettant de sélectionner des éléments HTML pour pouvoir par la suite les modifier, les animé, ou autre.

Avant de voir ces méthodes, quelques explications sont nécessaires à propos de l'objet **Windows** est sa propriété **document**

Document

Le **document** est une propriété de l'objet Windows il représente le **HTML** et on s'en sert pour modifier ou créer toute chose.

Il est impossible de lister toutes les méthodes et propriétés du document.

Méthodes de sélection de base

Voici un exemple de **HTML**, nous allons explorer différentes méthodes JavaScript pour sélectionner et agir sur ces éléments.

```
<div id="app">
  <div class="container">
    <section>
      <article>
        <h2>Titre de l'article</h2>
        <p>1er paragraphe</p>
        <p>2ème paragraphe</p>
        <p id="test">3ème paragraphe</p>
        <a href="#">lien</a>
        <ul>
          <li class="test"><a href="#">1ère li</a></li>
          <li><a href="#">2ème li</a></li>
          <li><a href="#">3ème li</a></li>
        </ul>
      </article>
    </section>
  </div>
</div>
```

getElementById

```
let element = document.getElementById('app');
console.log(element);
```

Cette méthode permet de sélectionnez un élément par son id, et elle retourne un élément HTML. On peut ensuite agir dessus pour modifier le CSS par exemple .

```
element.style.backgroundColor = 'lightgrey';
element.style.padding = '1em';
```

Méthodes de sélection de base

La méthode `querySelector` permet de sélectionner un élément, mais elle est beaucoup moins rigide que la précédente elle accepte tout type de sélecteur css.

```
<div id="app">
  <div class="container">
    <section>
      <article>
        <h2>Titre de l'article</h2>
        <p>1er paragraphe</p>
        <p>2ème paragraphe</p>
        <p id="test">3ème paragraphe</p>
        <a href="#">lien</a>
        <ul>
          <li class="test"><a href="#">1ère li</a></li>
          <li><a href="#">2ème li</a></li>
          <li><a href="#">3ème li</a></li>
        </ul>
      </article>
    </section>
  </div>
</div>
```

`querySelector()`

let `element` = document. `querySelectorAll ('p')`; Sélectionne tous les P sur la page
console.log(`element`);

let `element` = document. `querySelector('li:nth-child(2)')`; Sélectionne 2^{ème} ``
console.log(`element.innerText`);

On peut aussi se servir d'un élément pour sélectionner un autre.

let `element` = document. `querySelectorAll('ul')`;
Let `li` = `element.querySelector('li:first-child')` Selectionnes le premier ``
console.log(`li`);

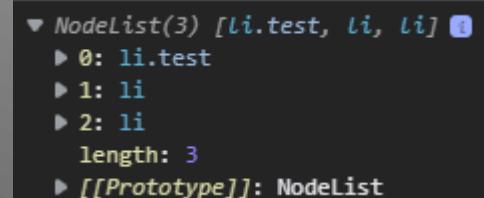
Méthodes de sélection de base

Cette méthode `querySelectorAll` permet de sélectionner un groupe d'éléments appelé nodelist. On peut sélectionner tous les éléments, elle accepte tout type de sélecteur `CSS` et également les pseudo sélecteurs

`querySelectorAll()`

```
<div id="app">
  <div class="container">
    <section>
      <article>
        <h2>Titre de l'article</h2>
        <p>1er paragraphe</p>
        <p>2ème paragraphe</p>
        <p id="test">3ème paragraphe</p>
        <a href="#">lien</a>
        <ul>
          <li class="test"><a href="#">1ère li</a></li>
          <li><a href="#">2ème li</a></li>
          <li><a href="#">3ème li</a></li>
        </ul>
      </article>
    </section>
  </div>
</div>
```

```
let element = document.querySelectorAll('li'); // Sélectionne tous les <li>
Let li = element.querySelector('li:first-child') Selectionnes le premier <li>
console.log(li);
```



```
let element = document.querySelectorAll(':hover'); // pseudo sélecteur .
console.log(element);
Sélectionnez les éléments avec un pseudo sélecteur vous sera utile que lorsque en géreras les événements
```

Méthodes de sélection de base

```
<div id="app">
  <div class="container">
    <section>
      <article>
        <h2>Titre de l'article</h2>
        <p>1er paragraphe</p>
        <p>2ème paragraphe</p>
        <p id="test">3ème paragraphe</p>
        <a href="#">lien</a>
        <ul>
          <li class="test"><a href="#">1ère li</a></li>
          <li><a href="#">2ème li</a></li>
          <li><a href="#">3ème li</a></li>
        </ul>
      </article>
    </section>
  </div>
</div>
```

querySelectorAll()

```
let element = document.querySelectorAll('ul > li:first-child'); // premier enfant de
```

```
<ul>
```

```
element.classList.add('yellow');
```

```
▼ <ul>
  ▼ <li class="test yellow"> == $0
    ::marker
    <a href="#">1ère li</a>
    </li>
  ▶ <li>...</li>
  ▶ <li>...</li>
</ul>
```

```
element.classList.remove('test');
```

```
▼ <ul>
  ▼ <li class="yellow"> == $0
    ::marker
    <a href="#">1ère li</a>
    </li>
  ▶ <li>...</li>
  ▶ <li>...</li>
</ul>
```

Méthodes de sélection de base

La méthode `toggle` est souvent utilisé pour cacher où afficher un élément.

Cette méthode ajoute une classe si l'élément sélectionné ne possède pas cette classe elle enlève la classe si l'élément la possède on peut imaginer cliquer un bouton et quand on le clic, on affiche où cache un élément.

```
/*CSS*/
.active {
    display: block;
}

// JS
a.classList.toggle('active');
```

Les events



les **events** sont des évènements qui se produisent quand on navigue sur une page, quand la souris bouge, quand on clique un élément ou quand on tape sur le clavier par exemple. Il existe plusieurs types d'évènements, les natifs et ceux qu'on crée. Pour l'instant, on va voir les events les plus courantes, qui se produisent quand la page web a finie de se charger, quand on tape sur le clavier et quand on agit avec la souris.

On ajoute ce qu'on appelle des **events listener** avec une méthode qui accepte deux arguments, le premier argument est l'évènement qu'on souhaite écouter, le second est une fonction dite 'callback' qui sera exécuté quand l'évènement se produira. La fonction de callback peut accepter optionnellement l'évènement. Ajoutons l'évènement suivant :

DOMContentLoaded

Cet évènement se produit quand la page web est chargée et indique que tous les éléments sont 'prêts' et qu'on peut agir sur la page avec le code javascript.

```
document.addEventListener('DOMContentLoaded', function(event) {  
    console.log('ready');  
    console.log(event);  
});
```

Voici une autre façon d'écrire un **eventListener**, en crée une fonction séparée et on la passe par la référence en Callback

```
function domReady(event) {  
    console.log('domReady');  
    console.log(event);  
}  
document.addEventListener('DOMContentLoaded', domReady);  
Il est judicieux d'exécuter Le javascript après qu'on ait reçu cet événement
```

Mouve events

Il y a également plusieurs types d'évènements concernant la souris : le clic, le double clic, les mouvements etc ect..

Voyant quelque un:

```
// clique gauche simple
document.getElementById("btn").addEventListener('click', function(event) {
    console.log("je clique")
    //On empêche le comportement par défaut de l'élément :
    //Si l'élément est un lien, il dirigera pas la ou le href indique : il ne fera rien
    event.preventDefault();
    // On log l'élément qu'on a cliqué :
    console.log(event.target);
});

// double clique :
document.getElementById("btnDouble").addEventListener('dblclick', function(event) {
    console.log("je double clique")
    event.preventDefault();
    console.log(event);
});

// mouvement de la souris
section_1.addEventListener('mousemove', function(event) {
    section_1.style.background = "red"
    console.log("changement de couleur")
});
```

Voici quelques évènements souris disponible : mousedown, mouseup, mouseover, mouseout, click, dblclick, contextmenu



Keyboard Events

Les événements du clavier peuvent être utilisés pour faire défiler les images d'une galerie de photos par exemple.

Différentes événements sont disponibles : Keydown, keyup.

Quant on console.log() l'événement, on se rend compte qu'on a différentes propriétés qu'on peut utiliser pour identifier la touche du clavier, ensuite on peut prévenir en fonctionnement par défaut et faire ce qu'on veut faire avec

<https://tutorial.eyehunts.com/js/javascript-keycode-events-keydown-keypress-and-keyup/>

```
document.addEventListener('keydown', function(event) {
    console.log(event);
    if(event.code === 'Space') {
        event.preventDefault(); // Empêcher le déroulement de la page
        console.log('On change la photo du slider');
    }
});
```



Switch

```
switch (expression) { une expression à comparer avec chacune des cases
    case valeur1:
        // Instructions à exécuter lorsque le résultat
        // de l'expression correspond à valeur1
        instructions1;
        [break];
    case valeur2:
        // Instructions à exécuter lorsque le résultat
        // de l'expression correspond à valeur2
        instructions 2;
        [break];
    ...
    case valeurN:
        // Instructions à exécuter lorsque le résultat
        // de l'expression à valeurN
        instructionsN;
        [break];
    default:
        // Instructions à exécuter lorsqu'aucune des valeurs
        // ne correspondent
        instructions_def;
        [break;]
}
```

Les timers

Vous l'avez bien compris maintenant JavaScript s'exécute sur le navigateur du client et permet d'interagir avec lui.

JavaScript va donc pouvoir détecter les évènements éventuels qui se produisent sur la page.

Pour le moment vous avez vu 2 types d'évènements :

les évènements lancés par le navigateur lui même :

j'ai fini de charger et lire le HTML (**DOMContentLoaded**),

j'ai fini de charger toute la page et ses dépendances (**load**),

etc...

les évènements lancés par le client (l'utilisateur) :

je clique sur un bouton ou une zone (**click**),

je survole une zone (**mouseover**),

je scrolle sur la page (**scroll**),

je soumet un formulaire (**submit**)

etc...

Nous pouvons donc interagir facilement avec ce qu'il se passe sur la page. Pourtant il nous manque une notion importante pour compléter ce panel d'évènements :

Comment déterminer le temps qui passe et s'en servir pour lancer des actions

Un exemple très simple de "**timer**", vous y êtes confronté très régulièrement sur Internet : au bout de 10 secondes une popup s'ouvre et vous propose de vous abonner à une Newsletter.

La gestion du temps en JavaScript va nous servir à bien d'autres choses que de la simple information ou communication agressive. Nous allons même y trouver ce qu'il y a de mieux, gérer des animations, temporiser des actions, et de façon plus complexe créer des jeux vidéos par exemple en utilisant toutes les fonctionnalités évènementielles de JavaScript !



SetTimeout et clearTimeout

Cette méthode va nous permettre de définir un intervalle en millisecondes avant le déclenchement d'une action.

```
let timeoutId = window.setTimeout(callBackFunction, [ delay, param1, param2, ...]);
```

Paramètres

callBackFunction : c'est la fonction qui sera appelée une fois le délai dépassé. On peut ici fournir un nom de fonction défini ailleurs dans notre code ou directement une fonction anonyme (comme pour toutes les fonctions évènementielles JavaScript qui appellent une fonction de callBack)

delay (optionnel) : le délai en millisecondes avant que la fonction de callBack ne soit appelée. Par défaut, ce paramètre vaut 0, la fonction est exécutée dès que possible. Astuce : 1000 ms = 1 s

param1, param2... (optionnels) : les paramètres (valeurs) qui seront passés à la fonction de callBack. Attention ces paramètres ne sont pas compatibles avec IE9 (Internet Explorer 9)

Valeur de retour

timeoutId : un identifiant unique fourni par Javascript si vous souhaitez retrouver et arrêter votre "timer" avant son exécution par exemple,

clearTimeout

Cette méthode va nous permettre d'arrêter le "timer" s'il n'a pas encore été déclenché.

```
window.clearTimeout(timeoutId);
```

SetTimeout et clearTimeout

Cette méthode va nous permettre de définir un intervalle en millisecondes avant le déclenchement d'une action.

```
let timeoutId = window.setTimeout(callBackFunction, [ delay, param1, param2, ...]);
```

Paramètres

timeoutId : un identifiant unique fourni par Javascript que nous récupérons au lancement de setTimeout.

Exemple : un message `alert` après 3 secondes

En utilisant une fonction anonyme directement dans l'appel de `setTimeout`

```
let alertId = window.setTimeout(  
    function(){window.alert('Cela fait 3 secondes que cette ligne Javascript a été lu par le navigateur !')},  
    3000);
```

En utilisant une fonction fonction de `callBackFunction` nommée

```
let alertId = window.setTimeout(alertInfo,3000);
```

```
function alertInfo() {  
    window.alert('Cela fait 3 secondes que cette ligne Javascript a été lu par le navigateur !');  
}
```



SetInterval et clearInterval

setInterval

Cette méthode va nous permettre de définir une action qui sera exécutée à intervalles réguliers.

```
let intervalID = window.setInterval(callBackFunction, [ delay, param1, param2, ...]);
```

Paramètres

callBackFunction : c'est la fonction qui sera appelée à tous les intervalles déterminés. On peut ici fournir un nom de fonction défini ailleurs dans notre code ou directement une fonction anonyme (comme pour toutes les fonctions évènementielles JavaScript qui appellent une fonction de callBack),

delay (optionnel) : le délai en millisecondes entre chaque intervalles d'exécution de la fonction de callBack. Par défaut, ce paramètre vaut 0, la fonction est exécutée dès que possible et continuellement. Astuce : 1000 ms = 1 s.

param1, param2... (optionnels) : les paramètres (valeurs) qui seront passés à la fonction de callBack. Attention ces paramètres ne sont pas compatibles avec IE9 (Internet Explorer 9).

Valeur de retour

timeoutID : un identifiant unique fourni par Javascript si vous souhaitez retrouver et arrêter votre "timer" à tout moment.

clearInterval

Cette méthode va nous permettre d'arrêter le "timer" à tout moment.

```
window.clearInterval(intervalID);
```

SetInterval et clearInterval

Exemple : afficher un compteur de secondes depuis le chargement complet de la page

Le code JavaScript du fichier main.js

'use strict'

```
// Initialisation du compteur
let timer = 0;

// Evènement qui détecte quand la page est intégralement chargée (ajout d'un gestionnaire d'évènement "load" sur la fenêtre "window")
window.addEventListener('load', function () {

    // Zone du DOM qui contient le compteur
    let timerDom = document.querySelector('#timer');

    // Execution d'une fonction de mise à jour du compteur toutes les secondes. Cette fonction reçoit en paramètre la zone du DOM à mettre à jour
    let intervalID = window.setInterval(updateTimer, 1000, timerDom);
});

/** Fonction qui met à jour le timer en ajoutant 1
 * @param {HTMLElement} objet du DOM où est contenu le texte du Timer
 */
function updateTimer(timerDom) {
    // on met à jour la zone HTML
    timerDom.innerText = ++timer;
}
```

Request animation frame

requestAnimationFrame et cancelAnimationFrame

requestAnimationFrame

Cette méthode va nous permettre de définir une action qui sera exécutée au rafraîchissement de l'écran.

```
let animationID = window.requestAnimationFrame(callBackFunction);
```

Paramètres

callBackFunction : c'est la fonction qui sera appelée au prochain rafraîchissement de l'écran. On peut ici fournir un nom de fonction défini ailleurs dans notre code ou directement une fonction anonyme (comme pour toutes les fonctions événementielles JavaScript qui appellent une fonction de callBack),

Valeur de retour

animationID : un identifiant unique fourni par Javascript si vous souhaitez retrouver et arrêter votre demande de rafraîchissement à tout moment.

cancelAnimationFrame

Cette méthode va nous permettre d'arrêter l'exécution du rafraîchissement à tout moment.

```
window.cancelAnimationFrame(animationID);
```

Paramètres

animationID : un identifiant unique fourni par Javascript que nous récupérons au lancement de requestAnimationFrame.

SetInterval et clearInterval

Mieux comprendre le fonctionnement : les FPS

Alors que setInterval et setTimeout fonctionnent avec un délai en millisecondes, requestAnimationFrame lui va être exécuté à chaque fois que le navigateur fait appel au rafraîchissement de l'écran auprès de la machine du client (rafraîchissement assuré en grande partie par le processeur graphique).

Cela signifie que :

la méthode requestAnimationFrame est plus adapté à la création d'animations fluides,

le rafraîchissement va dépendre de la machine client, en général un écran est rafraîchi 60 fois par seconde, on dit que l'on affiche 60 FPS (frames par seconde),

le rafraîchissement va dépendre aussi du travail demandé au navigateur mais aussi de sa capacité à ce moment là d'assurer le travail (capacité de l'ordinateur, navigateur plus ou moins utilisé et nombre d'onglets ouverts), les FPS sont donc variables comme dans les jeux vidéos, mais le rafraîchissement sera effectué quoi qu'il arrive,

le navigateur va optimiser et préparer le rafraîchissement, contrairement à un setInterval, il sait ce qu'il a à faire au prochain rafraîchissement et va le prioriser et l'optimiser,

le navigateur ne fera l'action de rafraîchissement que si l'onglet du navigateur comportant le script est actif, contrairement à setInterval et setTimeout qui continueront à être exécutés même si un autre onglet est actif (donc on optimise la charge du navigateur).

Alors tout ça à l'air magique mais il y a ici une chose que nous ne maîtrisons pas c'est le nombre de frames par seconde (FPS). Il nous faudra ralentir le taux de rafraîchissement si nous en avons besoin. L'exemple ci-dessous nous montre comment obtenir "facilement" la valeur des FPS !

```

// Initialisation des fps
let fps = 0;
// Zone du DOM qui contient le texte des FPS
let fpsDom;

// Tableau nous permettant de stocker le nombre d'appel au rafraîchissement et la valeur du temps en millisecondes
let times = [];

// Evènement qui détecte quand la page est intégralement chargée (ajout d'un gestionnaire d'évènement "load" sur la fenêtre "window")
window.addEventListener('load', function () {
    // Zone du DOM qui contient le texte des FPS
    fpsDom = document.querySelector('#fps');

    // on demande l'exécution d'une fonction lors du rafraîchissement
    window.requestAnimationFrame(updateFps);
});

function updateFps()
{
    // On indique au navigateur de rappeler notre fonction au prochain rafraîchissement
    window.requestAnimationFrame(updateFps);

    // On récupère le nombre de millisecondes (timestamp)
    const now = performance.now();

    // on va enlever les valeurs dans le tableau qui sont supérieurs à 1 seconde (on garde donc dans le tableau le temps de tous les appels s'ils ont été effectués dans la même seconde). Si le tableau comporte 60 éléments, on aura donc fait 60 appels en 1 seconde, donc 60 FPS !
    while (times.length > 0 && times[0] <= now - 1000) {
        times.shift();
    }

    // on ajoute au tableau le temps actuel en millisecondes
    times.push(now);

    // La longueur du tableau correspond donc au nombre de fois où nous avons appelé cette fonction en une seconde, donc à nos FPS
    fps = times.length;

    // on affiche les FPS dans l'HTML
    fpsDom.innerText = `${fps} fps`;
}

```



Canvas



C'est quoi Canvas ?

Une balise HTML et des librairies Javascript

```
<canvas width="800" height="600"></canvas>
```

Quelques petites choses à savoir :

- Canvas définit une zone graphique modifiable en JavaScript,
- La taille d'un Canvas, généralement en pixels, doit-être définie directement dans la balise HTML ou en JS mais pas en CSS (**sinon c'est flou**).
- Canvas peut-être utilisé pour créer des graphiques en 2D (Context2D) ou en 3D (WebGL), nous aborderons ici le dessin 2D,
- Canvas est normalement optimisé pour fonctionner avec le GPU (processeur graphique de l'ordinateur). Les animations y seront plus fluides et optimisées,
- javascript possède des Librairies (Context2D ou WebGL) pour modifier les graphiques présents dans la zone Canvas.
Par exemple y ajouter du texte, une image, un dessin, appliquer une transformation (rotation, translation), etc...

"Oui mais" on nous a appris à animer du HTML, pourquoi passer par Canvas.

Animer de l'HTML est une chose.

Ces animations servent à dynamiser la page, mais ne sont pas optimisées pour la performance.

Quelles soit en JS, en CSS ou avec un mélange des deux ces animations sont cosmétiques.

Si l'on souhaite faire un jeu en full HTML, c'est possible mais le rendu ne sera pas forcément adapté. Vous vous souvenez de notre balle animée avec requestAnimationFrame, dans certains navigateurs la balle est déformée.

Ce ne sera pas le cas avec Canvas !

C'est quoi Canvas ?

A quoi peut nous servir **Canvas** :

- Créer un jeu vidéo : Exemple Casse Brique
- Permettre la modification d'une zone graphique ou même un éditeur de photo comme Gimp ou Photoshop en ligne : Exemple Pixlr
- Permettre l'enregistrement d'une zone graphique sous la forme d'une image (GIF, JPG, PNG).



PRÉSENTATION JavaScript



PRÉSENTATION JavaScript

Les class

Déclaration de class

La syntaxe:

```
class CustomErrors { // }
```

A quoi sert une classe ?

Une classe permet de grouper au même endroit des morceaux de codes qui gèrent le même type de problèmes.

Une classe ***ne doit faire qu'une seule chose, elle n'a qu'une responsabilité***, ici on prendra comme exemple la gestion d'erreur.

Dans notre domaine, nous devons en permanence gérer la validité des formulaires

Pour cela, nous pouvons créer une classe qui ne gèrera que les erreurs et ajouter toutes les fonctions qui gèrent les erreurs dans cette classe.

Quand on ajoute un fonction dans une classe, on ne l'appelle plus fonction mais ***méthode***.

Voici la syntaxe pour ajouter une ***méthode*** dans une classe.

```
class CustomErrors {  
    recordErrors() { // }  
}
```

Déclaration de class

Tout comme pour une fonction 'normale', notre **méthode** peut accepter des arguments, et faire un travail avec.

```
class CustomErrors {  
    recordErrors(errors) { this.errors = errors; }  
}
```

Pour comprendre ce morceau de code, quelques explications sont nécessaires:

Le mot-clé:

this

Le mot-clé: **this** représente simplement l'objet ou instance issue de la classe.

Ainsi

this.errors

signifie qu'on veut accéder à une variable **errors** de notre classe, qu'on appelle une variable membre.

Grâce à ce mot-clé, on peut aussi appeler les méthodes de la classe.

Pour l'instant la classe ne sait pas qu'elle a une variable **errors**.

Pour la déclarer, il nous faut un constructeur:

```
class CustomErrors {  
    constructor() {  
        this.errors = {};  
    }  
    recordErrors(errors) {  
        this.errors = errors; }  
}
```

Exercice

Exercice 1 - Classe "Calculette"

Écrivez une classe nommée "Calculette" qui accepte 2 arguments lors de l'instanciation : nombre1 et nombre2 et qui les renseignent en tant que propriété.

La classe devra également être composée de ces 5 méthodes (qui n'auront pas d'arguments) :

additionner() : renvoie l'addition de nombre1 et nombre2

soustraire() : ...

multiplier() : ...

diviser() : ...

modulo() : ...

Exemple d'utilisation :

```
let calc = new Calculette(5, 7);
calc.additionner(); // doit afficher 12
```

Assurez-vous également que lors du changement d'un des nombre, ce dernier soit un Number. Sinon, levez une erreur :

```
calc.nombre1 = "6"; // doit afficher une erreur "Ceci n'est pas un nombre!"
L'utilisation d'un setter pour vérifier la valeur entrante vous sera utile ;)
```

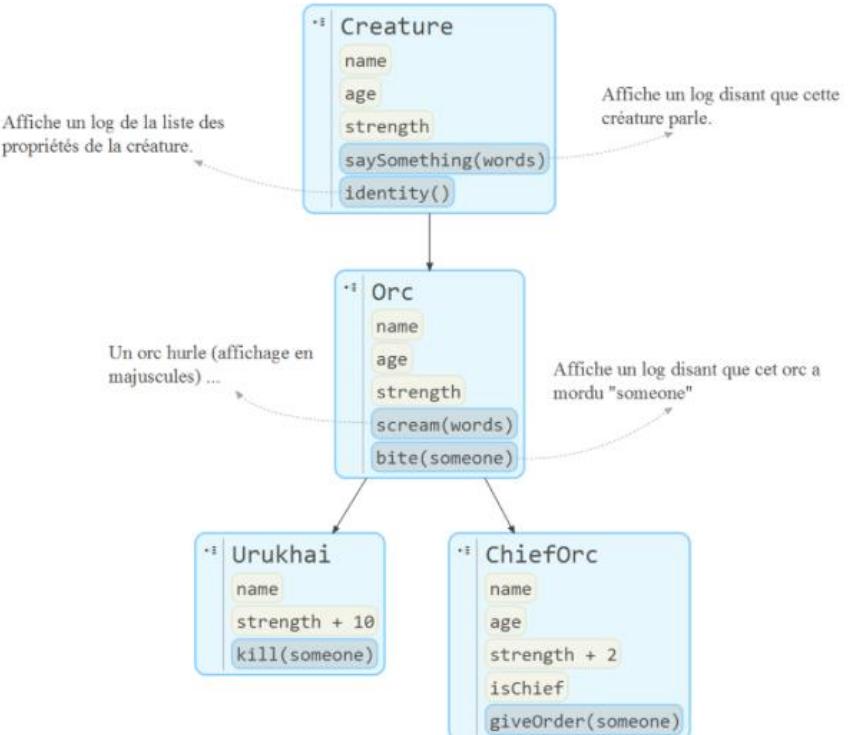
Bonus (si vous avez déjà fini) !

Écrire la même implémentation avec la syntaxe ES5 "à l'ancienne" (avec la function et le prototype) ;)

Exercice

1. Reprendre l'exercice sur les Orcs
2. À partir du fichier `Orcs.js` , créer 4 modules js contenant respectivement les 4 classes concernées. Chacun de ces modules devront ****exporter par défaut**** leur classe.
3. Créer un fichier `index.js` et `index.html`
4. Dans le HTML, faire appel au fichier principal `index.js` (avec le type "module")
5. Dans `index.js` , importer les 4 classes et placer ensuite le code de l'invocation

Exercice 2 - Orcs



Écrivez les classes **Creature**, **Orc**, **Urukhai** et **ChiefOrc** conformément au schéma ci-dessus, sachant que :

Urukhai et **ChiefOrc** héritent de **Orc**, qui elle-même hérite de **Creature**.

Actions :

- Une **Creature** peut faire : `saySomething()` et `identity()`
- Un **Orc** peut faire (en plus de `saySomething()` et `identity()`) : `scream()` et `bite()`
- Un **Urukhai** peut faire (en plus de `saySomething()`, `identity()`, `scream()` et `bite()`) : `kill()`
- Un **ChiefOrc** peut faire (en plus de `saySomething()`, `identity()`, `scream()` et `bite()`) : `giveOrder()`

Bonus (si vous avez déjà fini) !

Écrire la même implémentation avec la syntaxe ES5 "à l'ancienne" (avec la `function`, `prototype`, `call`, `Objet.create` et `constructor`) ;)

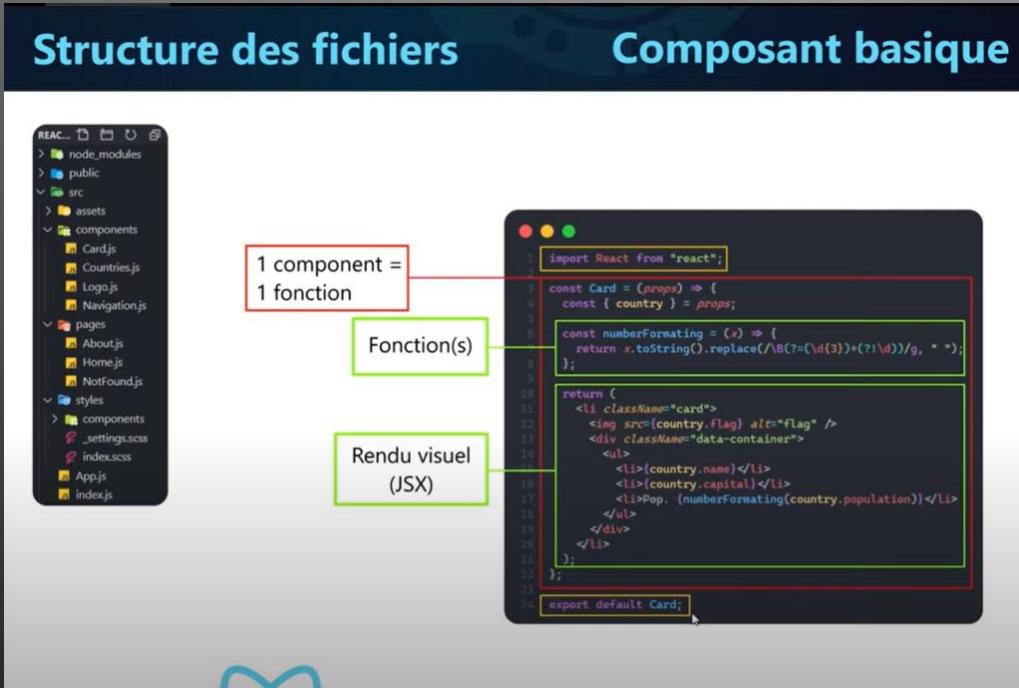
NPM Module

Installation, configuration



Explication

npx create-react-app mon-app



Installer React

Dans la console (se placer sur le dossier ou bureau)

✓ node -v

✓ npm -v

✓ npx create-react-app nom-projet

✓ cd nom-projet

✓ code .

Lancer projet : npm start

Reprendre projet : npm i

Compiler projet : npm run build

Bibliothèques utiles :
npm i -s react-router react-router-dom node-sass

NPX

C'est un exécuteur de paquets qui est inclus dans npm 5.2+

npx create-react-app mon-app



CMD : **terminale**

npm -v

Installation de Node Js : **version actuelle 14.17.6**

Npm install -g creact-react-app

cd nom-app

npm start : **lancement du sevrer**

Explication

```
npx create-react-app mon-app
```



classList

La propriété `classList` est utilisé pour ajouter ou enlever des classes CSS aux éléments sélectionnés.

```
let a = document.querySelector('ul > li:first-child a');
a.classList.add('yellow');
a.classList.remove('test');
```

Ici, on a ajouté simplement la classe yellow et on a enlevé la classe test.



Vue.js

Make Frontend Development Great Again!



Les frameworks sont incontournables dans le développement frontend.

La maîtrise de l'un d'eux représente donc un véritable atout pour votre carrière en développement frontend.

Il existe de nombreux frameworks.

Vue.js est l'un des plus populaires pour construire des applications web progressives.



Npm 1

Vue peut être utilisé pour se faciliter la tâche quand on développe des applications simple, mais aussi des 'Single Page Applications'
On peut utiliser Vue seul ou avec d'autres librairies.

Node-v

Npm -v

npm install -g @vue/cli

Vue create nom-du-projets

La façon la plus simple pour débuter avec Vue est d'intégrer un script depuis un CDN

```
<script src="https://cdn.jsdelivr.net/npm/vue/dist/vue.js"></script>>
```

Toutes les applications Vue commence en créant une nouvelle instance de Vue grâce à la fonction Vue:

```
new Vue(  
  { //options }  
);
```

La fonction Vue accepte un objet en argument, cet objet contiendra nos données, fonctions et bien d'autres, qui serviront à gérer le comportement des pages.

Installer l'extension officielle VueJs pour votre navigateur

[Vue.js devtools – Get this Extension for !\[\]\(1e3e06aa6b11a99464bdc92766bd0a7b_img.jpg\) Firefox \(en-US\) \(mozilla.org\)](#)

[Vue.js devtools - Chrome Web Store \(google.com\)](#)



Une application Vue est contenu dans une instance 'racine' (le root élément qu'on voit dans l'onglet Vue dans la console.) créée avec la fonction Vue, organisée de manière optionnelle en différents composants.

Chaque composant apparaitra comme un enfant de cette instance.

L'élément root est un élément HTML auquel on attache une instance de Vue.

Voici le code qui rend cette page dynamique.

```
1 import HeaderTemplate from './header.js';
2 import FooterTemplate from './footer.js';
3 new Vue({
4   el: '#app', // on attache l'instance de Vue à cet élément.
5   components: { HeaderTemplate, FooterTemplate }, // Le footer et le header sous forme de composants
6   data() {
7     return {
8       showVueIntro: false, // On commence avec le texte explicatif caché
9     };
10  },
11  computed: {
12    classes() {
13      return this.showVueIntro ? 'active' : 'inactive'; // On applique une classe CSS dynamiquement.
14    },
15  },
16 });
17 })
```

Ce code est certainement obscur pour l'instant mais vous allez voir au fur et à mesure de la semaine qu'il est en réalité très simple de travailler avec cette librairie.



Basic Data Binding 2

Vue.js recherche un élément par son ID, qu'on lui spécifie quand on l'instancie.

```
<div id="app"></div>
new Vue({
  el: '#app'
});
```

VueJS

L'id app sert d'ancre à Vue.js, on lui donne le nom qu'on veut, app ou root sont bien adaptés.

A partir de cet élémént 'racine', Vue.js sélectionne tous les éléments récursivement et recherche des propriétés, qu'on appelle des directives, qui rendront un élément réactif.

Examinons un exemple avec la directive v-model :

```
<body>
  <div id="app">
    <form>
      <label for="message">On teste la directive v-model</label>
      <input type="text" id="message" v-model="message">
    </form>
  </div>
```

HTML



Basic Data Binding 3

Avec le data object, on lie maintenant nos directives avec javascript pour les rendre réactives

```
new Vue({  
  el: '#app',  
  data: {  
    message: ''  
  }  
});
```

Avec ces courtes ligne de code, on a maintenant une propriété message réactive.

Si on tape du texte dans l'input, la propriété message prendra automatiquement la valeur de ce qu'on tape.

ON appelle ça *une liaison de données bidirectionnelle* ou *two-way data binding*.

Pour vérifier ça, on a plusieurs solutions, la directive v-text ou les 'moustaches':

```
<!-- la syntaxe moustache permet d'accéder aux propriétés de l'objet data -->

{{ message }}</p><!-- la directive v-text permet d'accéder aux propriétés de l'objet data -->

</p>


```

A noter :

Il existe deux façons de spécifier un élément root à Vue.js, voyons la deuxième :

```
new Vue({  
  data: {  
    message: '',  
    html: '<p>Testing awesomeness</p>'  
  }  
}).$mount('#app'); // on peut attacher le root element ici
```

Petit à petit, nous allons voir les directives et méthodes les plus utilisées de Vue.js

La syntaxe décrite ici est valable dans le cas où on utilise Vue.js sur une seule page. Ce n'est donc pas l'idéal, mais l'avantage est qu'il est très simple de démarrer pour s'entraîner.

Nous verrons plus tard la syntaxe recommandée dans le cadre des composants et d'une application faite avec Vue.js



List Rendering 4

Rendu de tableaux et objets avec Vue.js

Comment gérer les boucles avec Vue.js ?

Rendu d'un tableau dans une liste :

Ajoutons un tableau à notre data object

```
data() {
  return {
    arr: ['item', 'item 1', 'item2'],
  };
}
```

Et on tourne autour avec la directive v-for of

```
<ul>
  On extrait chaque element 'item' du tableau 'arr'
  <li v-for="item of arr">{{ item }}</li>
</ul>
```



List Rendering

Rendu d'un objet dans une table :

On ajoute un objet :

```
data() {
  return {
    obj: { name: 'John Doe', phone: '06-06-06-06-06' },
  };
}
```

Et on utilise la directive v-for in :

```
<table>
  <thead>
    <tr>
      <th>Nom</th>
      <th>Phone</th>
    </tr>
  </thead>
  <tbody>
    <tr>
      <!-- On extrait chaque propriétés 'item' de l'objet 'obj' -->
      <td v-for="item in obj">{{ item }}</td>
    </tr>
  </tbody>
</table>
```



List Rendering

Rendu d'un tableau composé d'objets dans une table :

On ajoute les données :

```
data() {
  return {
    arrOfObj: [
      { name: 'John Doe', phone: '06-06-06-06-06' },
      { name: 'Jane Doe', phone: '07-07-07-07-07' },
      { name: 'Jack Doe', phone: '08-08-08-08-08' },
    ],
  };
}
```

On met une directive v-for of pour extraire les objets du tableau, suivi d'une directive v-for in pour les objets :

```
<table>
  <thead>
    <tr>
      <th>Nom</th>
      <th>Phone</th>
    </tr>
  </thead>
  <tbody>
    <tr v-for="items of arrayOfObj">
      <td v-for="item in items">{{ item }}</td>
    </tr>
  </tbody>
</table>
```



List Rendering

Le script complet :

```
new Vue({
  el: '#app',

  data() {
    return {
      arr: ['item', 'item 1', 'item2'],
      obj: { name: 'John Doe', phone: '06-06-06-06-06' },
      arrOfObj: [
        { name: 'John Doe', phone: '06-06-06-06-06' },
        { name: 'Jane Doe', phone: '07-07-07-07-07' },
        { name: 'Jack Doe', phone: '08-08-08-08-08' },
      ],
    };
  },
});
```



Les Composants 5

Vue.js offre la possibilité de créer des 'components' réutilisable, c'est ici qu'on trouve la force de ce type de framework.

Un composant Vue.js est fait de markup HTML et de JavaScript et peut aussi comporter du css.

Il offre les mêmes possibilités qu'une instance de Vue. (méthodes, data object etc etc)

Quand on travaille dans une application faite avec Vue.js, on a accès à des fichiers avec l'extension .vue Dans ce cas la syntaxe est différente. Nous la verrons plus tard.

Dans un composant le data object **DOIT** être une fonction qui retourne un objet.

Un composant Vue.js doit être composé d'un seul root Element dans le template, dans le composant suivant, ce sera une

Un composant doit être nommé, ici on a créé un composant TaskList, on pourra s'en servir dans le HTML en le référençant ainsi:

```
<task-list />
```



Les Composants

```
52
53 //Voici le composant déclaré dans le fichier TaskList.js
54 // export du composant.      // Nom du composant
55 export default Vue.component('TaskList', [
56   // Le template HTML
57   template: `
58     <ul class="mt-1em">
59       <li>Faire de courses</li>
60       <li>Reviser les tableaux</li>
61       <li>Reviser les objets JS</li>
62       <li>Faire quelque requetes SQL</li>
63       <li>{{ message }}</li>
64     </ul>`, // Attention à la virgule.. facile de l'oublier..
65
66   // Dans un composant le data object ***DOIT*** être une fonction qui retourne un objet
67   data() {
68     return {
69       message:
70         'Dans un composant le data object ***DOIT*** être une fonction qui retourne un objet',
71     };
72   },
73 ]);
```

//Voici comment on l'importe :

```
import TaskList from './TaskList.js'; // Importer le composant

new Vue({
  el: '#app',
  components: { TaskList }, // NE pas oublier d'enregistrer le composant.
  data: {}
});
```

<task-list />



Event Handling 6

Etudions maintenant les event listeners : Tous les event listeners disponibles en JavaScript sont disponibles dans Vue.js, après tout, Vue.js c'est du JavaScript.
Débutons par voir le '`submit`' : Plusieurs syntaxes sont disponibles: `@submit` et `v-on:submit` On l'ajout simplement à un formulaire et on y ajoute le modificateur `.prevent` pour empêcher le comportement par défaut de formulaire.
On ajoute aussi le `v-model` au champs du formulaires.

```
<!-- le @submit.prevent pour e.preventDefault() -->
<form action="" @submit.prevent="createContact">
  <label for="name">Name</label>
  |  <input type="text" v-model="name" id="name">
  <label for="email">Email</label>
  |  <input type="email" v-model="email" id="email">
  <label for="phone">Phone</label>
  |  <input type="tel" v-model="phone" id="phone">
  <button type="submit">Submit</button>
</form>
```

On a donc écrit `@submit.prevent="createContact"`, ce qui signifie que nous allons appeler la méthode `createContact()` quand on envoie le formulaire.
Ecrivons maintenant notre objet data et la méthode `createContact()`, à l'aide de l'objet methods de Vue.js.

```
new Vue({
  el: '#app',
  data: {
    name: '',
    email: '',
    phone: ''
  },
  // L'objet methods permet d'écrire des fonctions, ces fonctions sont locales dans le contexte d'un 'component'
  methods: {
    createContact() {
      const contact = {
        name: this.name,
        email: this.email,
        phone: this.phone,
      };
      console.log(contact);
    },
  },
});
```

Et c'est tout. Grâce au data object, au v-model et au two-way data binding, les propriétés correspondantes se mettront automatiquement à jour avec la valeur qu'auront les champs du formulaire. Dès qu'on va envoyer le formulaire, le contact sera créé sans qu'on est rien d'autre à faire.



Conditions VueJs

```
//Équivalent de if, else if, else. Voici un exemple de code JavaScript, toujours l'objet data :  
new Vue({  
  el: '#app',  
  components: { HeaderTemplate, FooterTemplate },  
  data: {  
    persons: [  
      { name: 'John Doe', age: 18 },  
      { name: 'Jack Doe', age: 30 },  
      { name: 'Jane Doe', age: 45 },  
    ],  
  },  
});
```

```
<!-- Et voici le template HTML -->  
<ul>  
  <li v-for="person in persons" v-if="person.age === 18">  
    {{ person.name }}  
  </li>  
  <li v-else-if="person.age > 10 && person.age < 35">  
    {{ person.name }}  
  </li>  
  <li v-else>  
    {{ person.name }}  
  </li>  
</ul>
```

A ce stade de la formation, nous estimons que nous n'avons pas besoin de fournir des explications à propos de ces directives.



Computed

Une 'Computed Property' est un objet qui contient des méthodes.

C'est l'équivalent du Data Object mais dynamique.

Une 'Computed Property' est calculée de manière dynamique en fonction de la valeur d'autres propriétés.

```
new Vue({  
  el: '#app',  
  
  data() {  
    return {  
      tasks: [  
        { description: 'Faire des courses', done: true },  
        { description: 'Reviser', done: false },  
        { description: 'Lire mes emails', done: false },  
        { description: 'Faire le menage', done: true },  
      ],  
    };  
  },  
});
```

On a un liste de tâches, certaines sont effectuées, d'autres non. On veut les afficher sur la page selon leur statut, le problème est qu'avec une directive traditionnelle v-for, on ne peut pas les trier. C'est ici qu'interviennent les 'computed properties'.



Computed

```
//Le code, et les explications après :  
new Vue({  
  el: '#app',  
  components: { HeaderTemplate, FooterTemplate },  
  data() {  
    return {  
      isVisible: true,  
      tasks: [  
        { description: 'Faire des courses', done: true },  
        { description: 'Reviser', done: false },  
        { description: 'Lire mes emails', done: false },  
        { description: 'Faire le menage', done: true },  
      ],  
    };  
  },  
  computed: {  
    // Filtrage des tasks en fonction du statut  
    tasksNotDone() {  
      return this.tasks.filter(task => !task.done);  
    },  
  
    tasksDone() {  
      return this.tasks.filter(task => task.done);  
    },  
  },  
});
```

Petit rappel à propos de la fonction flèche : *Une fonction flèche sur une ligne effectue un retour implicite des données.*

L'argument de la fonction new Vue({}) est donc un objet, une des propriétés de cet objet est computed

Cette propriété accepte des méthodes et elle retourne le calcul de ces méthodes sous forme de propriétés de l'objet data.

Ainsi, on peut y accéder dans le template de la même manière dont on accède aux data object.

Un exemple rendra le tout plus intelligible : On se sert des computed properties pour afficher les tâches selon leur statut : Lisez le code attentivement, on accède aux computed properties directement en référençant le nom des méthodes.

```
<ul>  
  <li  
    v-for="task in tasksDone"  
    v-text="task.description"  
  </li>  
</ul>  
<h3>Tasks Not Done</h3>  
<ul>  
  <li  
    v-for="task in tasksNotDone"  
    v-text="task.description"  
  </li>  
</ul>  
<!-- On passe la méthode en référence, dans la directive v-for, et Vue.js fait le reste. --&gt;</pre>
```

